快乐的 Linux 命令行

Published
with GitBook



Table of Contents

本书是 The Linux Command Line 的中文版，除了可以在 gitbook 上面阅读之外，你也可以点击下面链接阅读本书：

- 中英文混合版
- 中文版 (PDF-2014-9-26)

本书的翻译由好奇猫团队发起，并社区协同努力 （翻译贡献者名单 ）

若发现错误，欢迎大家到 github 发 issue

引言

我想给大家讲个故事。

故事内容不是 Linus Torvalds 在1991年怎样写了 Linux 内核的第一个版本， 因为这些内容你可以在许多 Linux 书籍中读到。我也不想告诉你，更早之前，Richard Stallman 是如何开始 GNU 项目，设计了一个免费的类似 Unix 的操作系统。那也是一个很有意义的故事， 但大多数 Linux 书籍也讲到了它。

我想告诉大家一个你如何才能夺回计算机管理权的故事。

在20世纪70年代末，我刚开始和计算机打交道时，正进行着一场革命，那时的我还是一名大学生。 微处理器的发明，使普通老百姓（就如你和我）真正拥有一台计算机成为可能。今天， 人们难以想象，只有大企业和强大的政府才能够拥有计算机的世界，是怎样的一个世界。 简单说，你做不了多少事情。

今天，世界已经截然不同了。计算机遍布各个领域，从小手表到大型数据中心，及大小介于它们之间的每件东西。 除了随处可见的计算机之外，我们还有一个无处不在的连接所有计算机的网络。这已经开创了一个人们可以 自我营造和自由创作的奇妙的新时代，但在过去的二三十年里，一些事情仍然在发生着改变。一个大公司不断地把它的 管理权强加到世界上绝大多数的计算机上，并且决定你对计算机的操作权力。幸运地是，来自世界各地的人们， 正积极努力地做些事情来改变这种境况。通过编写自己的软件，他们一直在为维护电脑的管理权而战斗着。 他们建设着 Linux。

一提到 Linux， 许多人都会说到“自由”，但我不认为他们都知道“自由”的真正涵义。“自由”是一种权力， 它决定你的计算机能做什么，同时，只有知道计算机正在做什么你才能够拥有这种“自由”。 “自由”是指一台没有任何秘密的计算机，你可以从它那里了解一切，只要你用心的去寻找。

为什么使用命令行

你是否注意到，在电影中一个“超级黑客”坐在电脑前，从不摸一下鼠标， 就能够在30秒内侵入到超安全的军事计算机中。这是因为电影制片人意识到， 作为人类，我们应该本能地知道让计算机圆满完成工作的唯一途径，是用键盘来操纵计算机。

现在，大多数的计算机用户只是熟悉图形用户界面（GUI），并且产品供应商和此领域的学者会灌输给用户这样的思想， 命令行界面（CLI）是过去使用的一种很恐怖的东西。这就很不幸，因为一个好的命令行界面， 是用来和计算机进行交流沟通的非常有效的方式，正像人类社会使用文字互通信息一样。人们说，“图形用户界面让简单的任务更容易完成， 而命令行界面使完成复杂的任务成为可能”，到现在这句话仍然很正确。

因为 Linux 是以 Unix 家族的操作系统为模型写成的，所以它分享了 Unix 丰富的命令行工具。 Unix 在20世纪80年代初显赫一时(虽然，开发它在更早之前），结果，在普遍地使用图形界面之前， 开发了一种广泛的命令行界面。事实上，很多人选择 Linux（而不是其他的系统，比如说 Windows NT）是因为其强大的命令行界面，可以使“完成复杂的任务成为可能”。

这本书讲什么

这本书介绍如何生存在 Linux 命令行的世界。不像一些书籍仅仅涉及一个程序，比如像 shell 程序，bash。 这本书将试着向你传授如何与命令行界面友好相处。 它是怎样工作的？ 它能做什么？ 使用它的最好方法是什么？

这不是一本关于 **Linux** 系统管理的书。然而任何一个关于命令行的深入讨论，都一定会牵涉到 系统管理方面的内容，这本书仅仅提到一点儿管理方面的知识。但是这本书为读者准备好了学习更多内容的坚实基础， 毕竟要胜任系统管理工作也需要良好的命令行使用基本功。

这本书是围绕 **Linux** 而写的。许多书籍，为了扩大自身的影响力，会包含一些其它平台的知识， 比如 Unix, MacOS X 等。这样做，很多内容只能比较空泛的去讲了。另一方面， 这本书只研究了当代 Linux 发行版。虽然，对于使用其它类似于 Unix 系统的用户来说， 书中95%的内容是有用的，但这本书主要面向的对象是现代 Linux 命令行用户。

谁应该读这本书

这本书是为已经从其它平台移民到 Linux 系统的新手而写的。最有可能，你是使用某个 Windows 版本的高手。 或许是老板让你去管理一个 Linux 服务器，或许你只是一个桌面用户，厌倦了系统出现的各种 安全防御问题，而想要体验一下 Linux。很好，这里欢迎你们！

不过一般来说，对于 Linux 的启蒙教育，没有捷径可言。学习命令行富于挑战性，而且很费气力。 这并不是说 Linux 命令行很难学，而是它的知识量很大，不容易掌握。Linux 操作系统， 差不多有数以千计的命令可供用户操作。由此可见，要给自己提个醒，命令行可不是轻轻松松就能学好的。

另一方面，学习 Linux 命令行会让你受益匪浅，给你极大的回报。如果你认为， 现在你已经是一个高手了。别急，其实你还不知道什么才是真正的高手。不像其他一些计算机技能， 一段时间之后可能就被淘汰了，命令行知识却不会落伍，你今天所学到的，在十年以后， 都会有用处。命令行通过了时间的检验。

如果你没有编程经验，也不要担心，我会带你入门。

这本书的内容

这些材料是经过精心安排的，很像一位老师坐在你身旁，耐心地指导你。 许多作者用系统化的方式讲解这些材料，虽然从一个作者的角度考虑很有道理，但对于 Linux 新手来说， 他们可能会感到非常困惑。

另一个目的，是想让读者熟悉 Unix 的思维方式，这种思维方式不同于 Windows 的。在学习过程中， 我们会帮助你理解为什么某些命令会按照它们的方式工作，以及它们是怎样实现那样的工作方式的。 Linux 不仅是一款软件，也是 Unix 文化的一小部分，它有自己的语言和历史渊源。 同时，我也许会说些过激的话。

这本书共分为五部分，每一部分讲述了不同方面的命令行知识。除了第一部分， 也就是你正在阅读的这一部分，这本书还包括：

- 第二部分 — 学习 shell 开始探究命令行基本语言，包括命令组成结构， 文件系统浏览，编写命令行，查找命令帮助文档。
- 第三部分 — 配置文件及环境 讲述了如何编写配置文件，通过配置文件，用命令行来 操控计算机。
- 第四部分 — 常见任务及主要工具 探究了许多命令行经常执行的普通任务。类似于 Unix 的操作系统，例如 Linux, 包括许多经典的命令行程序，这些程序可以用来对数据进行 强大的操作。
- 第五部分 — 编写 Shell 脚本 介绍了 shell 编程，一个无可否认的基本技能，能够自动化许多 常见的计算任务，很容易学。通过学习 shell 编程，你会逐渐熟悉一些关于编程语言方面的概念， 这些概念也适用于其他的编程语言。

怎样阅读这本书

从头到尾的阅读。它并不是一本技术参考手册，实际上它更像一本故事书，有开头，过程，结尾。

前提条件

为了使用这本书，你需要安装 Linux 操作系统。你可以通过两种方式，来完成安装。

1. 在一台（不用很新）的电脑上安装 Linux。你选择哪个 Linux 发行版安装，是无关紧要的事。 虽然大多数人一开始选择安装 Ubuntu, Fedora, 或者 OpenSUSE。如果你拿不定主意，那就先试试 Ubuntu。 由于主机硬件配置不同，安装 Linux 时，你可能不费吹灰之力就装上了，也可能费了九牛二虎之力还装不上。 所以我建议，一台使用了几年的台式机，至少要有256M 的内存，6G 的硬盘可用空间。尽可能避免使用 笔记本电脑和无线网络，在 Linux 环境下，它们经常不能工作。
2. 使用“Live CD.” 许多 Linux 发行版都自带一个比较酷的功能，你可以直接从系统安装盘 CDROM 中运行 Linux， 而不必安装 Linux。开机进入 BIOS 设置界面，更改引导项，设置为“从 CDROM 启动”。

不管你怎样安装 Linux，为了练习书中介绍的知识，你需要有超级用户（管理员）权限。

当你在自己的电脑上安装了 Linux 系统之后，就开始一边阅读本书，一边练习吧。本书大部分内容 都可以自己动手练习，坐下来，敲入命令，体验一下吧。

> 为什么我不叫它*“GNU/Linux”*
>
> 在某些领域，把 Linux 操作系统称为 “GNU/Linux 操作系统”， 则政治立场正确。但 “Linux” 的问题是， 没有一个完全

> 正确的方式能命名它，因为它是由许许多多，分布在世界各地的贡献者们，合作开发而成的。 从技术层面讲，Linux 只是操作系统的内核名字，没别的含义。当然内核非常重要，因为有它， 操作系统才能运行起来，但它并不能构成一个完备的操作系统。
>
> Richard Stallman 是一个天才的哲学家，自由软件运动创始人，自由软件基金会创办者，他创建了 GNU 工程， 编写了第一版 GNU C 编译器（gcc）， 创立了 GNU 通用公共协议（the GPL)等等。 他坚持把 Linux 称为 “GNU/Linux”，为的是准确地反映 GNU 工程对 Linux 操作系统的贡献。 然而，尽管 GNU 项目早于 Linux 内核，项目的贡献应该得到极高的赞誉，但是把 GNU 用在 Linux 名字里， 这对其他为 Linux 的发展做出重大贡献的程序员来说，就不公平了。而且，我觉得要是叫也要叫 “Linux/GNU" 比较准确一些， 因为内核会先启动，其他一切都运行在内核之上。
>
> 在目前流行的用法中，“Linux” 指的是内核以及在一个典型的 Linux 发行版中所包含的所有免费及开源软件； 也就是说，整个 Linux 生态系统，不只有 GNU 项目软件。在操作系统商界，好像喜欢使用单个词的名字， 比如说 DOS, Windows, MacOS, Solaris, Irix, AIX. 所以我选择用流行的命名规则。然而， 如果你喜欢用 “GNU/Linux”，当你读这本书时，可以在脑子里搜索并替换 “Linux”。我不介意。

拓展阅读

Wikipedia 网站上有些介绍本章提到的名人的文章，以下是链接地址：

- http://en.wikipedia.org/wiki/Linux_Torvalds
- http://en.wikipedia.org/wiki/Richard_Stallman

介绍自由软件基金会及 GNU 项目的网站和文章：

- http://en.wikipedia.org/wiki/Free_Software_Foundation
- http://www.fsf.org
- http://www.gnu.org

Richard Stallman 用了大量的文字来叙述“GNU/Linux”的命名问题，可以浏览以下网页：

- http://www.gnu.org/gnu/why-gnu-linux.html
- http://www.gnu.org/gnu/gnu-linux-faq.html#tools

什么是 shell

一说到命令行，我们真正指的是 shell。shell 就是一个程序，它接受从键盘输入的命令， 然后把命令传递给操作系统去执行。几乎所有的 Linux 发行版都提供一个名为 bash 的程序， bash 是 shell 的一种，来自于 GNU 项目。“bash”是“Bourne Again SHell”的首字母缩写， 所指的是这样一个事实，bash 是 sh 的增强版，sh 是最初 Unix 的 shell 程序，由 Steve Bourne 写成。

终端仿真器

当使用图形用户界面时，我们需要另一个叫做终端仿真器的程序，去和 shell 交互。 浏览一下我们的桌面菜单，我们可能会找到一个。虽然在菜单里它可能都 被简单地称为“terminal”，但是 KDE 用的是 konsole 程序, 而 GNOME 则使用 gnome-terminal。 还有其他一些终端仿真器可供 Linux 使用，但基本上，它们都是为了完成同样的事情， 让我们能访问 shell。也许，你会喜欢上这个或那个终端，由于它所附加的一系列花俏功能。

第一次按键

好，开始吧。启动终端仿真器！一旦它运行起来，我们应该能够看到一行类似下面文字的输出：

```
[me@linuxbox ~]$
```

这叫做 shell 提示符，当 shell 准备好了去接受输入时，它就会出现。然而， 它可能会以各种各样的面孔显示，这则取决于不同的 Linux 发行版， 它通常包括你的用户名@主机名，紧接着当前工作目录（稍后会有更多介绍）和一个美元符号。

如果提示符的最后一个字符是“#”, 而不是“$”, 那么这个终端会话就有超级用户权限。 这意味着，我们或者是以根用户的身份登录，或者是我们选择的终端仿真器提供超级用户（管理员）权限。

假定到目前为止，所有事情都进行顺利，那我们试着打字吧。在提示符下敲入 一些乱七八糟的无用数据，如下所示：

```
[me@linuxbox ~]$ kaekfjaeifj
```

因为这个命令没有任何意义，所以 shell 会提示错误信息，并让我们再试一下：

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

命令历史

如果按下上箭头按键，我们会看到刚才输入的命令“kaekfjaeifj”重新出现在提示符之后。 这就叫做命令历史。许多 Linux 发行版默认保存最后输入的500个命令。 按下下箭头按键，先前输入的命令就消失了。

移动光标

可借助上箭头按键，来获得上次输入的命令。现在试着使用左右箭头按键。 看一下怎样把光标定位到命令行的任意位置？通过使用箭头按键，使编辑命令变得轻松些。

关于鼠标和光标

虽然，shell 是和键盘打交道的，但你也可以在终端仿真器里使用鼠标。X 窗口系统 （使 GUI 工作的底层引擎）内建了一种机制，支持快速拷贝和粘贴技巧。 如果你想高亮一些文本，可以按下鼠标左键，沿着文本拖动鼠标（或者双击一个单词），那么这些高亮的文本就被拷贝到了一个由 X 管理的缓冲区里面。然后按下鼠标中键， 这些文本就被粘贴到光标所在的位置。试试看。

注意： 不要在一个终端窗口里，使用 Ctrl-c 和 Ctrl-v 快捷键，来执行拷贝和粘贴操作。 它们不起作用。对于 shell 来说，这

些控制代码有着不同的含义，它们被赋值，早于 Microsoft Windows 许多年。

你的图形桌面环境（像 KDE 或 GNOME），努力想和 Windows 一样，可能会把它的聚焦策略 设置成“单击聚焦”。这意味着，为了让窗口聚焦（变得活跃）你需要单击它。 这与“聚焦跟随着鼠标”的传统 X 行为不同，传统 X 行为是指只要把鼠标移动到一个窗口的上方， 这个窗口就成为活动窗口。这个窗口不会成为前端窗口，直到你单击它，但它能接受输入。 设置聚焦策略为“聚焦跟随着鼠标”，可以使拷贝和粘贴技巧更有益。尝试一下。 给它一个机会，我想你会喜欢上它的。在窗口管理器的配置程序中，你会找到这个设置。

试试运行一些简单命令

现在，我们学习了怎样输入命令，那我们执行一些简单的命令吧。第一个命令是 date。 这个命令显示系统当前时间和日期。

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2007
```

一个相关联的命令，cal，它默认显示当前月份的日历。

```
[me@linuxbox ~]$ cal
October 2007
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

查看磁盘剩余空间的数量，输入 df:

```
[me@linuxbox ~]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5012392   9949716  34% /
/dev/sda5             59631908  26545424  30008432  47% /home
/dev/sda1               147764     17370    122765  13% /boot
tmpfs                   256856         0    256856   0% /dev/shm
```

同样地，显示空闲内存的数量，输入命令 free。

```
[me@linuxbox ~]$ free
total       used       free     shared    buffers     cached
Mem:       2059676     846456    1213220          0
44028      360568
-/+ buffers/cache:     441860    1617816
Swap:      1042428          0    1042428
```

结束终端会话

我们可以终止一个终端会话，通过关闭终端仿真器窗口，或者是在 shell 提示符下输入 exit 命令：

```
[me@linuxbox ~]$ exit
```

幕后控制台

即使，终端仿真器没有运行，几个终端会话仍然在后台运行着。它们叫做虚拟终端 或者是虚拟控制台。在大多数 Linux 发行版中，这些终端会话都可以访问， 按下 Ctrl-Alt-F1 到 Ctrl-Alt-F6 访问不同的虚拟终端。当一个会话被访问的时候， 它会显示登录提示框，我们需要输入用户名和密码。从一个虚拟控制台转换到另一个， 按下 Alt 和 F1-F6。返回图形桌面，按下 Alt-F7。

拓展阅读

- 想了解更多关于 Steve Bourne 的故事，Bourne Shell 之父，读一下这篇文章：

  http://en.wikipedia.org/wiki/Steve_Bourne

- 这是一篇关于在计算机领域里，shells 概念的文章：

  http://en.wikipedia.org/wiki/Shell_(computing)

文件系统中跳转

我们需要学习的第一件事（除了打字之外）是如何在 Linux 文件系统中跳转。 在这一章节中，我们将介绍以下命令：

- pwd — 打印出当前工作目录名
- cd — 更改目录
- ls — 列出目录内容

理解文件系统树

类似于 Windows，一个“类 Unix” 的操作系统，比如说 Linux，以分层目录结构来组织所有文件。 这就意味着所有文件组成了一棵树型目录（有时候在其它系统中叫做文件夹）， 这个目录树可能包含文件和其它的目录。文件系统中的第一级目录称为根目录。 根目录包含文件和子目录，子目录包含更多的文件和子目录，依此类推。

不同于 Windows 的是，Windows 每个存储设备都有一个独自的文件系统，类似于 Unix 的操作系统， 比如 Linux， 总是有一个单一的文件系统树，不管有多少个磁盘或者存储设备连接到计算机上。 根据系统管理员的兴致，存储设备连接到（或着更精确些，是挂载到）目录树的各个节点上。 系统管理员负责维护系统安全。

当前工作目录

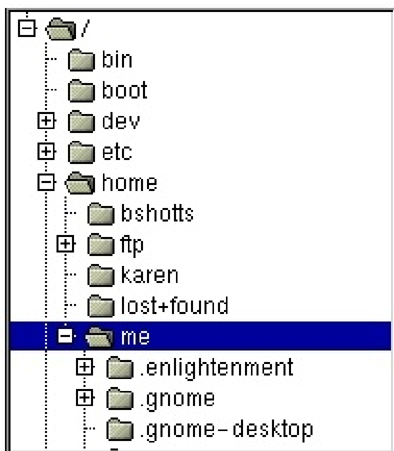


图 1: 显示在文件管理器中的系统文件树

大多数人都可能熟悉图形文件管理器，它描述了文件系统树的结构，正如图1所示。 注意通常，这是一棵倒置的树，也就是说，树根在最上面，而各个枝干在下面展开。

然而，命令行没有图片，所以我们需要考虑用不同的方法，在文件系统树中跳转。

把文件系统想象成一个迷宫形状，就像一棵倒立的大树，我们站在迷宫的中间位置。 在任意时刻，我们处于一个目录里面，我们能看到这个目录包含的所有文件， 以及通往上面目录（父目录）的路径，和下面的各个子目录。我们所在的目录则称为当前工作目录。我们使用 pwd（打印工作目录）命令，来显示当前工作目录。

```
[me@linuxbox ~]$ pwd
/home/me
```

当我首次登录系统后，（或者启动终端仿真器会话后）， 当前工作目录是我的主目录。 每个用户都有他自己的主目录，当用户以普通用户的身份操控系统时，主目录是唯一 允许用户编写文件的地方。

列出目录内容

列出一个目录包含的文件及子目录，使用 ls 命令。

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

实际上，用 ls 命令可以列出任一个目录的内容，而不只是当前工作目录的内容。 ls 命令还能完成许多有趣的事情。在下一章节，我们将介绍更多关于 ls 的知识。

更改当前工作目录

要更改工作目录（此刻，我们站在树形迷宫里面），我们用 cd 命令。输入 cd, 然后输入你想要的工作目录的路径名，就能实现愿望。路径名就是沿着目录树的分支 到达想要的目录，期间所经过的路线。路径名可通过两种方式来指定，一个是绝对路径， 另一个是相对路径。首先处理绝对路径。

绝对路径

绝对路径开始于根目录，紧跟着目录树的一个个分支，一直到达期望的目录或文件。 例如，你的系统中有一个目录，大多数系统程序都安装在这个目录下。这个目录的 路径名是 /usr/bin。它意味着从根目录（用开头的"/"表示）开始，有一个叫 "usr" 的 目录包含了目录 "bin"。

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Listing of many, many files ...
```

我们把工作目录转到 /usr/bin 目录下，里面装满了文件。注意 shell 提示符是怎样改变的。 为了方便，通常设置提示符自动显示工作目录名。

相对路径

绝对路径从根目录开始，直到它的目的地，而相对路径开始于工作目录。 一对特殊符号来表示相对位置，在文件系统树中。这对特殊符号是 "." (点) 和 ".." (点点)。

符号 "." 指的是工作目录，".." 指的是工作目录的父目录。下面的例子说明怎样使用它。 再次更改工作目录到 /usr/bin：

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

好的，比方说更改工作目录到 /usr/bin 的父目录 /usr。可以通过两种方法来实现。或者使用绝对路径名：

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

或者， 使用相对路径：

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

两种不同的方法，一样的结果。我们应该选哪一个呢？ 输入量最少的那个。

同样地，从目录 /usr/ 到 /usr/bin 也有两种途径。或者使用绝对路径：

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

或者，用相对路径：

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

有一件很重要的事，我必须指出来。在几乎所有的情况下，你可以省略"./"。它是隐含地。输入：

```
[me@linuxbox usr]$ cd bin
```

实现相同的效果，如果不指定一个文件的目录，那它的工作目录会被假定为当前工作目录。

有用的快捷键

在表3-1中，列举出了一些快速改变当前工作目录的有效方法。

表3-1: cd 快捷键

快捷键	运行结果
cd	更改工作目录到主目录。
cd -	更改工作目录到先前的工作目录。
cd ~user_name	更改工作目录到用户主目录。例如, cd ~bob 会更改工作目录到用户“bob”的主目录。

> 关于文件名的重要规则
>
> 1. 以 "." 字符开头的文件名是隐藏文件。这仅表示，ls 命令不能列出它们， 除非使用 ls -a 命令。当你创建帐号后，几个配置帐号的隐藏文件被放置在 你的主目录下。稍后，我们会仔细研究一些隐藏文件，来定制你的系统环境。另外，一些应用程序也会把它们的配置文件以隐藏文件的形式放在你的主目录下面。
>
> 2. 文件名和命令名是大小写敏感的。文件名 File1 和 file1 是指两个不同的文件名。
>
> 3. Linux 没有“文件扩展名”的概念，不像其它一些系统。可以用你喜欢的任何名字 来给文件起名。文件内容或用途由其它方法来决定。虽然类似 Unix 的操作系统， 不用文件扩展名来决定文件的内容或用途，但是应用程序会。
>
> 4. 虽然 Linux 支持长文件名，文件名可能包含空格，标点符号，但标点符号仅限 使用 “.”，“－”，下划线。最重要的是，不要在文件名中使用空格。如果你想表示词与 词间的空格，用下划线字符来代替。过些时候，你会感激自己这样做。

探究操作系统

既然我们已经知道了如何在文件系统中跳转，是时候开始 Linux 操作系统之旅了。在开始之前，我们先学习一些对研究 Linux 系统有帮助的命令。

- ls — 列出目录内容
- file — 确定文件类型
- less — 浏览文件内容

ls 乐趣

有充分的理由证明，ls 可能是用户最常使用的命令。通过它，我们可以知道目录的内容，以及各种各样重要文件和目录的 属性。正如我们所知道的，只简单的输入 ls 就能看到在当前目录下所包含的文件和子目录列表。

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Publica Templates Videos
```

除了当前工作目录以外，也可以列出指定目录的内容，就像这样：

```
me@linuxbox ~]$ ls /usr
bin games   kerberos    libexec  sbin   src
etc include lib         local    share  tmp
```

甚至可以列出多个指定目录的内容。在这个例子中，将会列出用户主目录（用字符“~”代表）和/usr 目录的内容：

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop  Documents  Music  Pictures  Public  Templates  Videos

/usr:
bin  games     kerberos  libexec  sbin   src
etc  include   lib       local    share  tmp
```

我们也可以改变输出格式，来得到更多的细节：

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Desktop
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Documents
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Music
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Pictures
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Public
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Templates
drwxrwxr-x 2  me  me  4096  2007-10-26  17:20  Videos
```

使用 ls 命令的“-l”选项，则结果以长模式输出。

选项和参数

我们将学习一个非常重要的知识点，大多数命令是如何工作的。命令名经常会带有一个或多个用来更正命令行为的选项， 更进一步，选项后面会带有一个或多个参数，这些参数是命令作用的对象。所以大多数命令看起来像这样：

```
command -options arguments
```

大多数命令使用的选项，是由一个中划线加上一个字符组成，例如，“-l”，但是许多命令，包括来自于 GNU 项目的命令，也支持长选项，长选项由两个中划线加上一个字组成。当然， 许多命令也允许把多个短选项串在一起使用。下面这个例子，ls 命令有两个选项， “l” 选项产生长格式输出，“t”选项按文件修改时间的先后来排序。

```
[me@linuxbox ~]$ ls -lt
```

加上长选项 “--reverse”，则结果会以相反的顺序输出：

```
[me@linuxbox ~]$ ls -lt --reverse
```

ls 命令有大量的选项。表4-1列出了最常使用的选项。

表 4-1: ls 命令选项

选项	长选项	描述
-a	--all	列出所有文件，甚至包括文件名以圆点开头的隐藏文件。
-d	--directory	通常，如果指定了目录名，ls 命令会列出这个目录中的内容，而不是目录本身。把这个选项与-l 选项结合使用，可以看到所指定目录的详细信息，而不是目录中的内容。
-F	--classify	这个选项会在每个所列出的名字后面加上一个指示符。例如，如果名字是 目录名，则会加上一个'/'字符。
-h	--human-readable	以长格式列出。以人们可读的格式，而不是以字节数来显示文件的大小。
-l		以长格式显示结果。
-r	--reverse	以相反的顺序来显示结果。通常，ls 命令的输出结果按照字母升序排列。
-S		命令输出结果按照文件大小来排序。
-t		按照修改时间来排序。

深入研究长格式输出

正如我们先前知道的，“-l”选项导致 ls 的输出结果以长格式输出。这种格式包含大量的有用信息。下面的例子目录来自 于 Ubuntu 系统：

```
-rw-r--r-- 1 root root 3576296 2007-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2007-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2007-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2007-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2007-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2007-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2007-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2007-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2007-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root  453764 2007-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2007-04-03 11:05 ubuntu Sax.ogg
```

选一个文件，来看一下各个输出字段的含义：

表 4-2: ls 长格式列表的字段

字段	含义
-rw-r--r--	对于文件的访问权限。第一个字符指明文件类型。在不同类型之间， 开头的“－”说明是一个普通文件，“d”表明是一个目录。其后三个字符是文件所有者的 访问权限，再其后的三个字符是文件所属组中成员的访问权限，最后三个字符是其他所 有人的访问权限。这个字段的完整含义将在第十章讨论。
1	文件的硬链接数目。参考随后讨论的关于链接的内容。
root	文件属主的用户名。

root	文件所属用户组的名字。
32059	以字节数表示的文件大小。
2007-04-03 11:05	上次修改文件的时间和日期。
oo-cd-cover.odf	文件名。

确定文件类型

随着探究操作系统的进行，知道文件包含的内容是很有用的。我们将用 file 命令来确定文件的类型。我们之前讨论过， 在 Linux 系统中，并不要求文件名来反映文件的内容。然而，一个类似 “picture.jpg” 的文件名，我们会期望它包含 JPEG 压缩图像，但 Linux 却不这样要求它。可以这样调用 file 命令：

```
file filename
```

当调用 file 命令后，file 命令会打印出文件内容的简单描述。例如：

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

有许多类型的文件。事实上，在类似于 Unix 操作系统中比如说 Linux，有个普遍的观念就是“任何东西都是一个文件”。 随着课程的进行，我们将会明白这句话的真谛。

虽然系统中许多文件格式是熟悉的，例如 MP3和 JPEG 文件，但也有一些文件格式比较含蓄，极少数文件相当陌生。

用 **less** 浏览文件内容

less 命令是一个用来浏览文本文件的程序。纵观 Linux 系统，有许多人类可读的文本文件。less 程序为我们检查文本文件 提供了方便。

> 什么是“文本”
>
> 在计算机中，有许多方法可以表达信息。所有的方法都涉及到，在信息与一些数字之间确立一种关系，而这些数字可以 用来表达信息。毕竟，计算机只能理解数字，这样所有的数据都被转换成数值表示法。
>
> 有些数值表达法非常复杂（例如压缩的视频文件），而其它的就相当简单。最早也是最简单的一种表达法，叫做 ASCII 文本。ASCII（发音是 "As-Key"）是美国信息交换标准码的简称。这是一个简单的编码方法，它首先 被用在电传打字机上，用来实现键盘字符到数字的映射。
>
> 文本是简单的字符与数字之间的一对一映射。它非常紧凑。五十个字符的文本翻译成五十个字节的数据。文本只是包含 简单的字符到数字的映射，理解这点很重要。它和一些文字处理器文档不一样，比如说由微软和 OpenOffice.org 文档 编辑器创建的文件。这些文件，和简单的 ASCII 文件形成鲜明对比，它们包含许多非文本元素，来描述它的结构和格式。 普通的 ASCII 文件，只包含字符本身，和一些基本的控制符，像制表符，回车符及换行符。纵观 Linux 系统，许多文件 以文本格式存储，也有许多 Linux 工具来处理文本文件。甚至 Windows 也承认这种文件格式的重要性。著名的 NOTEPAD.EXE 程序就是一个 ASCII 文本文件编辑器。

为什么我们要查看文本文件呢？因为许多包含系统设置的文件（叫做配置文件），是以文本格式存储的，阅读它们 可以更深入的了解系统是如何工作的。另外，许多系统所用到的实际程序（叫做脚本）也是以这种格式存储的。 在随后的章节里，我们将要学习怎样编辑文本文件，为的是修改系统设置，还要学习编写自己的脚本文件，但现在我们只是看看它们的内容而已。

less 命令是这样使用的：

```
less filename
```

一旦运行起来，less 程序允许你前后滚动文件。例如，要查看一个定义了系统中全部用户身份的文件，输入以下命令：

```
[me@linuxbox ~]$ less /etc/passwd
```

一旦 less 程序运行起来，我们就能浏览文件内容了。如果文件内容多于一页，那么我们可以上下滚动文件。按下“q”键，退出 less 程序。

下表列出了 less 程序最常使用的键盘命令。

表 4-3: less 命令

命令	行为
Page UP or b	向后翻滚一页
Page Down or space	向前翻动一页
UP Arrow	向前移动一行
Down Arrow	向后移动一行
G	移动到最后一行
1G or g	移动到开头一行
/charaters	向前查找指定的字符串
n	向前查找下一个出现的字符串，这个字符串是之前所指定查找的
h	显示帮助屏幕
q	退出 less 程序

less 就是 more（禅语：色即是空）

less 程序是早期 Unix 程序 more 的改进版。“less” 这个名字，对习语 “less is more” 开了个玩笑，这个习语是现代主义建筑师和设计者的座右铭。

less 属于"页面调度器"程序类，这些程序允许通过页方式，在一页中轻松地浏览长长的文本文档。然而 more 程序只能向前分页浏览，而 less 程序允许前后分页浏览，它还有很多其它的特性。

旅行指南

Linux 系统中，文件系统布局与类似 Unix 系统的文件布局很相似。实际上，一个已经发布的标准，叫做 Linux 文件系统层次标准，详细说明了这种设计模式。不是所有Linux发行版都根据这个标准，但 大多数都是。

下一步，我们将在文件系统中游玩，来了解 Linux 系统的工作原理。这会给你一个温习跳转命令的机会。我们会发现很多有趣的文件都是普通的可读文本。将开始旅行，做做以下练习：

1. cd 到给定目录
2. 列出目录内容 ls -l
3. 如果看到一个有趣的文件，用 file 命令确定文件内容
4. 如果文件看起来像文本，试着用 less 命令浏览它

记得复制和粘贴技巧！如果你正在使用鼠标，双击文件名，来复制它，然后按下鼠标中键，粘贴文件名到命令行中。

在系统中游玩时，不要害怕粘花惹草。普通用户是很难把东西弄乱的。那是系统管理员的工作！如果一个命令抱怨一些事情，不要管它，尽管去玩别的东西。花一些时间四处走走。系统是我们自己的，尽情地探究吧。记住在 Linux 中，没有秘密存在！表4-4仅仅列出了一些我们可以浏览的目录。闲暇时试试看！

表 4-4: Linux 系统中的目录

目录	评论
/	根目录，万物起源。
/bin	包含系统启动和运行所必须的二进制程序。
/boot	包含 Linux 内核，最初的 RMA 磁盘映像（系统启动时，由驱动程序所需），和 启动加载程序。 有趣的文件： • /boot/grub/grub.conf or menu.lst， 被用来配置启动加载程序。 • /boot/vmlinuz，Linux 内核。
/dev	这是一个包含设备结点的特殊目录。“一切都是文件”，也适用于设备。 在这个目录里，内核维护着它支持的设备。
/etc	这个目录包含所有系统层面的配置文件。它也包含一系列的 shell 脚本， 在系统启动时，这些脚本会运行每个系统服务。这个目录中的任何文件应该是可读的文本文件。 有意思的文件：虽然/etc 目录中的任何文件都有趣，但这里只列出了一些我一直喜欢的文件： • /etc/crontab， 定义自动运行的任务。 • /etc/fstab，包含存储设备的列表，以及与他们相关的挂载点。 • /etc/passwd，包含用户帐号列表。
/home	在通常的配置环境下，系统会在/home 下，给每个用户分配一个目录。普通只能 在他们自己的目录下创建文件。这个限制保护系统免受错误的用户活动破坏。
/lib	包含核心系统程序所需的库文件。这些文件与 Windows 中的动态链接库相似。
/lost+found	每个使用 Linux 文件系统的格式化分区或设备，例如 ext3文件系统， 都会有这个目录。当部分恢复一个损坏的文件系统时，会用到这个目录。除非文件系统 真正的损坏了，那么这个目录会是个空目录。
/media	在现在的 Linux 系统中，/media 目录会包含可移除媒体设备的挂载点， 例如 USB 驱动器，CD-ROMs 等等。这些设备连接到计算机之后，会自动地挂载到这个目录结点下。
/mnt	在早些的 Linux 系统中，/mnt 目录包含可移除设备的挂载点。
/opt	这个/opt 目录被用来安装“可选的”软件。这个主要用来存储可能 安装在系统中的商业软件产品。
/proc	这个/proc 目录很特殊。从存储在硬盘上的文件的意义上说，它不是真正的文件系统。 反而，它是一个由 Linux 内核维护的虚拟文件系统。它所包含的文件是内核的窥视孔。这些文件是可读的， 它们会告诉你内核是怎样监管计算机的。
/root	root 帐户的主目录。
/sbin	这个目录包含“系统”二进制文件。它们是完成重大系统任务的程序，通常为超级用户保留。
/tmp	这个/tmp 目录，是用来存储由各种程序创建的临时文件的地方。一些配置，导致系统每次 重新启动时，都会清空这个目录。
/usr	在 Linux 系统中，/usr 目录可能是最大的一个。它包含普通用户所需要的所有程序和文件。
/usr/bin	/usr/bin 目录包含系统安装的可执行程序。通常，这个目录会包含许多程序。
/usr/lib	包含由/usr/bin 目录中的程序所用的共享库。
/usr/local	这个/usr/local 目录，是非系统发行版自带，却打算让系统使用的程序的安装目录。 通常，由源码编译的程序会安装在/usr/local/bin 目录下。新安装的 Linux 系统中，会存在这个目录， 但却是空目录，直到系统管理员放些东西到它里面。
/usr/sbin	包含许多系统管理程序。
/usr/share	/usr/share 目录包含许多由/usr/bin 目录中的程序使用的共享数据。 其中包括像默认的配置文件，图标，桌面背景，音频文件等等。
/usr/share/doc	大多数安装在系统中的软件包会包含一些文档。在/usr/share/doc 目录下， 我们可以找到按照软件包分类的文档。
/var	除了/tmp 和/home 目录之外，相对来说，目前我们看到的目录是静态的，这是说， 它们的内容不会改变。/var 目录是可能需要改动的文件存储的地方。各种数据库，假脱机文件， 用户邮件等等，都驻扎在这里。
	这个/var/log 目录包含日志文件，各种系统活动的记录。这些文件非常重要，并且 应该时时监测它

/var/log	们。其中最重要的一个文件是/var/log/messages。注意，为了系统安全，在一些系统中， 你必须是超级用户才能查看这些日志文件。

符号链接

在我们到处查看时，我们可能会看到一个目录，列出像这样的一条信息：

```
lrwxrwxrwx 1 root root 11 2007-08-11 07:34 libc.so.6 -> libc-2.6.so
```

注意看，为何这条信息第一个字符是“l”，并且有两个文件名呢？ 这是一个特殊文件，叫做符号链接（也称为软链接或者 symlink ）。 在大多数”类 Unix“ 系统中，有可能一个文件被多个文件名所指向。虽然这种特性的意义并不明显，但它真地很有用。

描绘一下这样的情景：一个程序要求使用某个包含在名为“foo”文件中的共享资源，但是“foo”经常改变版本号。 这样，在文件名中包含版本号，会是一个好主意，因此管理员或者其它相关方，会知道安装了哪个“foo”版本。 这又会导致一个问题。如果我们更改了共享资源的名字，那么我们必须跟踪每个可能使用了 这个共享资源的程序，当每次这个资源的新版本被安装后，都要让使用了它的程序去寻找新的资源名。 这听起来很没趣。

这就是符号链接存在至今的原因。比方说，我们安装了文件 “foo” 的 2.6 版本，它的 文件名是 “foo-2.6”，然后创建了叫做 “foo” 的符号链接，这个符号链接指向 “foo-2.6”。 这意味着，当一个程序打开文件 “foo” 时，它实际上是打开文件 “foo-2.6”。 现在，每个人都很高兴。依赖于 “foo” 文件的程序能找到这个文件，并且我们能知道安装了哪个文件版本。 当升级到 “foo-2.7” 版本的时候，仅添加这个文件到文件系统中，删除符号链接 “foo”， 创建一个指向新版本的符号链接。这不仅解决了版本升级问题，而且还允许在系统中保存两个不同的文件版本。 假想 “foo-2.7” 有个错误（该死的开发者！），那我们得回到原来的版本。 一样的操作，我们只需要删除指向新版本的符号链接，然后创建指向旧版本的符号链接就可以了。

在上面列出的目录（来自于 Fedora 的 /lib 目录）展示了一个叫做 “libc.so.6” 的符号链接，这个符号链接指向一个 叫做 “libc-2.6.so” 的共享库文件。这意味着，寻找文件 “libc.so.6” 的程序，实际上得到是文件 “libc-2.6.so”。 在下一章节，我们将学习如何建立符号链接。

硬链接

讨论到链接问题，我们需要提一下，还有一种链接类型，叫做硬链接。硬链接同样允许文件有多个名字， 但是硬链接以不同的方法来创建多个文件名。在下一章中，我们会谈到更多符号链接与硬链接之间的差异问题。

拓展阅读

- 完整的 Linux 文件系统层次体系标准可通过以下链接找到：

  http://www.pathname.com/fhs/

操作文件和目录

此时此刻，我们已经准备好了做些真正的工作！这一章节将会介绍以下命令：

- cp — 复制文件和目录
- mv — 移动/重命名文件和目录
- mkdir — 创建目录
- rm — 删除文件和目录
- ln — 创建硬链接和符号链接

这五个命令属于最常使用的 Linux 命令之列。它们用来操作文件和目录。

现在，坦诚地说，用图形文件管理器来完成一些由这些命令执行的任务会更容易些。使用文件管理器， 我们可以把文件从一个目录拖放到另一个目录，剪贴和粘贴文件，删除文件等等。那么， 为什么还使用早期的命令行程序呢？

答案是命令行程序，功能强大灵活。虽然图形文件管理器能轻松地实现简单的文件操作，但是对于 复杂的文件操作任务，则使用命令行程序比较容易完成。例如，怎样复制一个目录下的 HTML 文件 到另一个目录，但这些 HTML 文件不存在于目标目录，或者是文件版本新于目标目录里的文件？ 要完成这个任务，使用文件管理器相当难，使用命令行相当容易：

```
cp -u *.html destination
```

通配符

在开始使用命令之前，我们需要介绍一个使命令行如此强大的 shell 特性。因为 shell 频繁地使用 文件名，shell 提供了特殊字符来帮助你快速指定一组文件名。这些特殊字符叫做通配符。 使用通配符（也以文件名代换著称）允许你依据字符类型来选择文件名。下表列出这些通配符 以及它们所选择的对象：

表5－1：通配符

通配符	意义
*	匹配任意多个字符（包括零个或一个）
?	匹配任意一个字符（不包括零个）
[characters]	匹配任意一个属于字符集中的字符
[!characters]	匹配任意一个不是字符集中的字符
[[:class:]]	匹配任意一个属于指定字符类中的字符

表5-2列出了最常使用的字符类：

表5－2：普遍使用的字符类

字符类	意义
[:alnum:]	匹配任意一个字母或数字
[:alpha:]	匹配任意一个字母
[:digit:]	匹配任意一个数字
[:lower:]	匹配任意一个小写字母
[:upper]	匹配任意一个大写字母

借助通配符，为文件名构建非常复杂的选择标准成为可能。下面是一些类型匹配的范例:

表5－3：通配符范例

模式	匹配对象
	所有文件
g	文件名以“g”开头的文件
b.*txt*	以"b"开头，中间有零个或任意多个字符，并以".txt"结尾的文件
Data???	以“Data”开头，其后紧接着3个字符的文件
[abc]	文件名以"a","b",或"c"开头的文件
BACKUP.[0-9][0-9][0-9]	以"BACKUP."开头，并紧接着3个数字的文件
[[:upper:]]	以大写字母开头的文件
[![:digit:]]	不以数字开头的文件
*[[:lower:]123]	文件名以小写字母结尾，或以 “1”，“2”，或 “3” 结尾的文件

接受文件名作为参数的任何命令，都可以使用通配符，我们会在第八章更深入的谈到这个知识点。

> 字符范围
>
> 如果你用过别的类似 Unix 系统的操作环境，或者是读过这方面的书籍，你可能遇到过[A-Z]或 [a-z]形式的字符范围表示法。这些都是传统的 Unix 表示法，并且在早期的 Linux 版本中仍有效。 虽然它们仍然起作用，但是你必须小心地使用它们，因为它们不会产生你期望的输出结果，除非 你合理地配置它们。从现在开始，你应该避免使用它们，并且用字符类来代替它们。
>
> 通配符在 GUI 中也有效
>
> 通配符非常重要，不仅因为它们经常用在命令行中，而且一些图形文件管理器也支持它们。
>
> - 在 Nautilus (GNOME 文件管理器）中，可以通过 Edit/Select 模式菜单项来选择文件。 输入一个用通配符表示的文件选择模式后，那么当前所浏览的目录中，所匹配的文件名 就会高亮显示。
> - 在 Dolphin 和 Konqueror（KDE 文件管理器）中，可以在地址栏中直接输入通配符。例如，如果你 想查看目录 /usr/bin 中，所有以小写字母 "u" 开头的文件，在地址栏中敲入 "/usr/bin/u*"，则 文件管理器会显示匹配的结果。
>
> 最初源于命令行界面中的想法，在图形界面中也适用。这就是使 Linux 桌面系统 如此强大的众多原因中的一个。

mkdir 命令是用来创建目录的。它这样工作：

```
mkdir directory...
```

注意表示法: 在描述一个命令时（如上所示），当有三个圆点跟在一个命令的参数后面， 这意味着那个参数可以重复，就像这样：

```
mkdir dir1
```

会创建一个名为"dir1"的目录，而

```
mkdir dir1 dir2 dir3
```

会创建三个目录，名为 dir1, dir2, dir3。

cp — 复制文件和目录

cp 命令，复制文件或者目录。它有两种使用方法：

```
cp item1 item2
```

复制单个文件或目录"item1"到文件或目录"item2"，和：

```
cp item... directory
```

复制多个项目（文件或目录）到一个目录下。

有用的选项和实例

这里列举了 cp 命令一些有用的选项（短选项和等效的长选项）：

表5－4：cp 选项

选项	意义
-a, --archive	复制文件和目录，以及它们的属性，包括所有权和权限。 通常，复本具有用户所操作文件的默认属性。
-i, --interactive	在重写已存在文件之前，提示用户确认。如果这个选项不指定， cp 命令会默认重写文件。
-r, --recursive	递归地复制目录及目录中的内容。当复制目录时， 需要这个选项（或者-a 选项）。
-u, --update	当把文件从一个目录复制到另一个目录时，仅复制 目标目录中不存在的文件，或者是文件内容新于目标目录中已经存在的文件。
-v, --verbose	显示翔实的命令操作信息

表5－5： cp 实例

命令	运行结果
cp file1 file2	复制文件 file1 内容到文件 file2。如果 file2 已经存在，file2 的内容会被 file1 的 内容重写。如果 file2 不存在，则会创建 file2。
cp -i file1 file2	这条命令和上面的命令一样，除了如果文件 file2 存在的话，在文件 file2 被重写之前，会提示用户确认信息。
cp file1 file2 dir1	复制文件 file1 和文件 file2 到目录 dir1。目录 dir1 必须存在。
cp dir1/* dir2	使用一个通配符，在目录 dir1 中的所有文件都被复制到目录 dir2 中。 dir2 必须已经存在。
cp -r dir1 dir2	复制目录 dir1中的内容到目录 dir2。如果目录 dir2 不存在， 创建目录 dir2，操作完成后，目录 dir2 中的内容和 dir1 中的一样。 如果目录 dir2 存在，则目录 dir1(和目录中的内容)将会被复制到 dir2 中。

mv — 移动和重命名文件

mv 命令可以执行文件移动和文件命名任务，这依赖于你怎样使用它。任何一种 情况下，完成操作之后，原来的文件名不再存在。mv 使用方法与 cp 很相像：

```
mv item1 item2
```

把文件或目录 “item1” 移动或重命名为 “item2”, 或者：

```
mv item... directory
```

把一个或多个条目从一个目录移动到另一个目录中。

有用的选项和实例

mv 与 cp 共享了很多一样的选项：

表5－6：mv 选项

选项	意义
-i --interactive	在重写一个已经存在的文件之前，提示用户确认信息。 如果不指定这个选项，**mv** 命令会默认重写文件内容。
-u --update	当把文件从一个目录移动另一个目录时，只是移动不存在的文件， 或者文件内容新于目标目录相对应文件的内容。
-v --verbose	当操作 mv 命令时，显示翔实的操作信息。

表5－7 mv 实例

mv file1 file2	移动 file1 到 file2。如果 **file2** 存在，它的内容会被 **file1** 的内容重写。 如果 file2 不存在，则创建 file2。 每种情况下，**file1** 不再存在。
mv -i file1 file2	除了如果 file2 存在的话，在 file2 被重写之前，用户会得到 提示信息外，这个和上面的选项一样。
mv file1 file2 dir1	移动 file1 和 file2 到目录 dir1中。dir1 必须已经存在。
mv dir1 dir2	如果目录 dir2 不存在，创建目录 dir2， 并且移动目录 dir1 的内容到目录 dir2 中，同时删除目录 dir1。 如果目录 dir2 存在，移动目录 dir1（及它的内容）到目录 dir2。

rm 命令用来移除（删除）文件和目录：

```
rm item...
```

"item"代表一个或多个文件或目录。

有用的选项和实例

下表是一些普遍使用的 rm 选项：

表5－8： rm 选项

选项	意义
-i, --interactive	在删除已存在的文件前，提示用户确认信息。如果不指定这个选项，**rm** 会默默地删除文件
-r, --recursive	递归地删除文件，这意味着，如果要删除一个目录，而此目录 又包含子目录，那么子目录也会被删除。要删除一个目录，必须指定这个选项。
-f, --force	忽视不存在的文件，不显示提示信息。这选项颠覆了"--interactive"选项。
-v, --verbose	在执行 rm 命令时，显示翔实的操作信息。

表 5－9： rm 实例

命令	运行结果
rm file1	默默地删除文件
rm -i file1	除了在删除文件之前，提示用户确认信息之外，和上面的命令作用一样。
rm -r file1 dir1	删除文件 file1, 目录 dir1，及 dir1中的内容。
rm -rf file1 dir1	同上，除了如果文件 file1，或目录 dir1不存在的话，rm 仍会继续执行。

> 小心 *rm!*
>
> 类似于 Unix 的操作系统，比如说 Linux，没有复原命令。一旦你用 rm 删除了一些东西， 它就消失了。Linux 假定你很聪明，你知道你在做什么。

> 尤其要小心通配符。思考一下这个经典的例子。假如说，你只想删除一个目录中的 HTML 文件。输入： rm *.html
>
> 这是正确的，如果你不小心在 "*" 和 ".html" 之间多输入了一个空格，就像这样：
>
> 这个 rm 命令会删除目录中的所有文件，还会抱怨没有文件叫做 ".html"。
>
> 小贴士。 无论什么时候，rm 命令用到通配符（除了仔细检查输入的内容外！）， 用 ls 命令来测试通配符。这会让你看到要删除的文件列表。然后按下上箭头按键，重新调用 刚刚执行的命令，用 rm 替换 ls。

ln — 创建链接

ln 命令即可创建硬链接，也可以创建符号链接。可以用其中一种方法来使用它：

```
ln file link
```

创建硬链接，和：

```
ln -s item link
```

创建符号链接，"item"可以是一个文件或是一个目录。

硬链接

硬链接和符号链接比起来，硬链接是最初 Unix 创建链接的方式，而符号链接更加现代。 在默认情况下，每个文件有一个硬链接，这个硬链接给文件起名字。当我们创建一个 硬链接以后，就为文件创建了一个额外的目录条目。硬链接有两个重要局限性：

1. 一个硬链接不能关联它所在文件系统之外的文件。这是说一个链接不能关联 与链接本身不在同一个磁盘分区上的文件。
2. 一个硬链接不能关联一个目录。

一个硬链接和文件本身没有什么区别。不像符号链接，当你列出一个包含硬链接的目录 内容时，你会看到没有特殊的链接指示说明。当一个硬链接被删除时，这个链接 被删除，但是文件本身的内容仍然存在（这是说，它所占的磁盘空间不会被重新分配）， 直到所有关联这个文件的链接都删除掉。知道硬链接很重要，因为你可能有时 会遇到它们，但现在实际中更喜欢使用符号链接，下一步我们会讨论符号链接。

符号链接

创建符号链接是为了克服硬链接的局限性。符号链接生效，是通过创建一个 特殊类型的文件，这个文件包含一个关联文件或目录的文本指针。在这一方面， 它们和 Windows 的快捷方式差不多，当然，符号链接早于 Windows 的快捷方式 很多年;-)

一个符号链接指向一个文件，而且这个符号链接本身与其它的符号链接几乎没有区别。 例如，如果你往一个符号链接里面写入东西，那么相关联的文件也被写入。然而， 当你删除一个符号链接时，只有这个链接被删除，而不是文件自身。如果删除这个文件 早于文件的符号链接，这个链接仍然存在，但是不指向任何东西。在这种情况下， 这个链接被称为坏链接。在许多实现中，ls 命令会以不同的颜色展示坏链接，比如说 红色，来显示它们的存在。

关于链接的概念，看起来很迷惑，但不要胆怯。我们将要试着练习 这些命令，希望，它变得清晰起来。

创建游戏场（实战演习）

下面我们将要做些真正的文件操作，让我们先建立一个安全地带， 来玩一下文件操作命令。首先，我们需要一个工作目录。在我们的 主目录下创建一个叫做"playground"的目录。

创建目录

mkdir 命令被用来创建目录。首先确定我们在我们的主目录下，来创建 playground 目录， 然后创建这个新目录：

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

为了让我们的游戏场更加有趣，在 playground 目录下创建一对目录 ，分别叫做"dir1"和"dir2"。更改我们的当前工作目录到 playground，然后 执行 mkdir 命令：

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

注意到 mkdir 命令可以接受多个参数，它允许我们用一个命令来创建这两个 目录。

复制文件

下一步，让我们得到一些数据到我们的游戏场中。通过复制一个文件来实现目的。 使用 cp 命令，我们从/etc 目录复制 passwd 文件到当前工作目录下：

```
[me@linuxbox playground]$ cp /etc/passwd .
```

注意：我们怎样使用当前工作目录的快捷方式，命令末尾的单个圆点。如果我们执行 ls 命令， 可以看到我们的文件：

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2  me  me   4096 2008-01-10 16:40 dir1
drwxrwxr-x 2  me  me   4096 2008-01-10 16:40 dir2
-rw-r--r-- 1  me  me   1650 2008-01-10 16:07 passwd
```

现在，仅仅是为了高兴，重复操作复制命令，使用"-v"选项（唠叨），看一个它的作用：

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

cp 命令再一次执行了复制操作，但是这次显示了一条简洁的信息，指明它 进行了什么操作。注意，cp 没有警告，就重写了第一次复制的文件。这是一个案例， cp 假定你知道你的所作所为。为了得到警示信息，在命令中包含"-i"选项：

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

响应命令提示信息，输入"y"，文件就会被重写，其它的字符（例如，"n"） 会导致 cp 命令不理会文件。

移动和重命名文件

现在，"passwd"这个名字，看起来不怎么有趣，这是个游戏场，所以我们给它改个名字：

```
[me@linuxbox playground]$ mv passwd fun
```

让我们来传送 fun 文件，通过移动重命名的文件到各个子目录， 然后再把它移回到当前目录：

```
[me@linuxbox playground]$ mv fun dir1
```

首先，把 fun 文件移动目录 dir1 中，然后：

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

再把 fun 文件从 dir1 移到目录 dir2, 然后：

```
[me@linuxbox playground]$ mv dir2/fun .
```

最后，再把 fun 文件带回到当前工作目录。下一步，来看看移动目录的效果。 首先，我们先移动我们的数据文件到 dir1 目录：

```
[me@linuxbox playground]$ mv fun dir1
```

然后移动 dir1 到 dir2 目录，用 ls 来确认执行结果:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2008-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2008-01-10 16:33 fun
```

注意：因为目录 dir2 已经存在，mv 命令移动 dir1 到 dir2 目录。如果 dir2 不存在， mv 会重新命名 dir1 为 dir2。最后，把所有的东西放回原处。

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

创建硬链接

现在，我们试着创建链接。首先是硬链接。我们创建一些关联我们 数据文件的链接：

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

所以现在，我们有四个文件"fun"的实例。看一下目录 playground 中的内容：

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me  me 4096 2008-01-14 16:17 dir1
drwxrwxr-x 2 me  me 4096 2008-01-14 16:17 dir2
-rw-r--r-- 4 me  me 1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me  me 1650 2008-01-10 16:33 fun-hard
```

注意到一件事，列表中，文件 fun 和 fun-hard 的第二个字段是"4"，这个数字 是文件"fun"的硬链接数目。你要记得一个文件至少有一个硬链接，因为文件 名就是由链接创建的。所以，我们怎样知道实际上 fun 和 fun-hard 是一样的文件呢？ 在这个例子里，ls 不是很有用。虽然我们能够看到 fun 和 fun-hard 文件大小一样 （第五字段），但我们的列表没有提供可靠的信息来确定（这两个文件一样）。 为了解决这个问题，我们更深入的研究一下。

当考虑到硬链接，想象文件是由两部分组成：数据部分包含文件的内容， 名字部分包含文件的名字，这样可以帮助理解。当我们创建了文件的硬链接， 实际上，我们给文件添加了额外的名字，这些名字都涉及一样的数据内容。 系统分配了一系列的

盘块给所谓的索引节点，它和文件名字想关联。因此每个硬 链接都关系到一个具体的索引节点，这个节点包含了文件的内容。

ls 命令有一种方法，来展示（文件索引节点）的信息。在命令中加上"-i"选项：

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me  me 4096  2008-01-14  16:17  dir1
12353540 drwxrwxr-x 2 me  me 4096  2008-01-14  16:17  dir2
12353538 -rw-r--r-- 4 me  me 1650  2008-01-10  16:33  fun
12353538 -rw-r--r-- 4 me  me 1650  2008-01-10  16:33  fun-hard
```

在这个版本的列表中，第一字段表示文件索引节点号，正如我们所见到的， fun 和 fun-hard 共享一样的索引节点号，这就证实这两个文件是一样的文件。

创建符号链接

建立符号链接的目的是为了克服硬链接的两个缺点：硬链接不能跨越物理设备， 硬链接不能关联目录，只能是文件。符号链接是文件的特殊类型，它包含一个指向 目标文件或目录的文本指针。

符号链接的建立过程相似于创建硬链接：

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

第一个实例相当直接，在 ln 命令中， 简单地加上"-s"选项就可以创建一个符号链接， 而不是一个硬链接。下面两个例子又是怎样呢？ 记住，当我们创建一个符号链接 的时候，会建立一个目标文件在哪里和符号链接有关联的文本描述。如果我们看看 ls 命令的输出结果，比较容易理解。

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me  me 1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me  me    6 2008-01-15 15:17 fun-sym -> ../fun
```

目录 dir1 中，fun-sym 的列表说明了它是一个符号链接，通过在第一字段中的首字符"l" 可知，并且它还指向"../fun"，也是正确的。相对于 fun-sym 的存储位置，fun 在它的 上一个目录。同时注意，符号链接文件的长度是6， 这是字符串"../fun"所包含的字符数， 而不是符号链接所指向的文件长度。

当建立符号链接时，你即可以使用绝对路径名：

```
ln -s /home/me/playground/fun dir1/fun-sym
```

也可用相对路径名，正如前面例题所展示的。使用相对路径名更令人满意， 因为它允许一个包含符号链接的目录重命名或移动，而不会破坏链接。

除了普通文件，符号链接也能关联目录：

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
...省略
```

移动文件和目录

正如我们之前讨论的，rm 命令被用来删除文件和目录。我们将要使用它 来清理一下我们的游戏场。首先，删除一个硬链接：

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
...省略
```

结果不出所料。文件 fun-hard 消失了，文件 fun 的链接数从4减到3，正如 目录列表第二字段所示。下一步，我们会删除文件 fun，仅为了娱乐，我们会包含"-i" 选项，看一个它的作用：

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

在提示符下输入"y"，删除文件。让我们看一下 ls 的输出结果。注意，fun-sym 发生了 什么事? 因为它是一个符号链接，指向已经不存在的文件，链接已经坏了：

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me  me     4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me  me        4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me     4096 2008-01-15 15:17 dir2
lrwxrwxrwx 1 me  me        3 2008-01-15 15:15 fun-sym -> fun
```

大多数 Linux 的发行版本配置 ls 显示损坏的链接。在 Fedora 系统中，坏的链接以闪烁的 红色文本显示！损坏链接的出现，并不危险，但是相当混乱。如果我们试着使用 损坏的链接，会看到以下情况：

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

稍微清理一下现场。删除符号链接：

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me  me    4096 2008-01-15 15:17 dir1
drwxrwxr-x 2 me  me    4096 2008-01-15 15:17 dir2
```

对于符号链接，有一点值得记住，执行的大多数文件操作是针对链接的对象，而不是链接本身。 而 rm 命令是个特例。当你删除链接的时候，删除链接本身，而不是链接的对象。

最后，我们将删除我们的游戏场。为了完成这个工作，我们将返回到 我们的主目录，然后用 rm 命令加上选项(-r)，来删除目录 playground， 和目录下的所有内容，包括子目录：

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

> 用 *GUI* 来创建符号链接
>
> 文件管理器 GNOME 和 KDE 都提供了一个简单而且自动化的方法来创建符号链接。 在 GNOME 里面，当拖动文件时，同时按下 Ctrl+Shift 按键会创建一个链接，而不是 复制（或移动）文件。在 KDE 中，无论什么时候放下一个文件，会弹出一个小菜单， 这个菜单会提供复制，移动，或创建链接文件选项。

总结

在这一章中，我们已经研究了许多基础知识。我们得花费一些时间来全面的理解。 反复练习 playground 例题，直到你觉得它有意义。能够良好的理解基本文件操作 命令和通配符，非常重要。空闲时，通过添加文件和目录来拓展 playground 练习， 使用通配符来为各种各样的操作命令指定文件。关于链接的概念，在刚开始接触 时会觉得有点迷惑，花些时间来学习它们是怎样工作的。它们能成为真正的救星。

使用命令

在这之前，我们已经知道了一系列神秘的命令，每个命令都有自己奇妙的 选项和参数。在这一章中，我们将试图去掉一些神秘性，甚至创建我们自己 的命令。这一章将介绍以下命令：

- type – 说明怎样解释一个命令名
- which – 显示会执行哪个可执行程序
- man – 显示命令手册页
- apropos – 显示一系列适合的命令
- info – 显示命令 info
- whatis – 显示一个命令的简洁描述
- alias – 创建命令别名

到底什么是命令？

命令可以是下面四种形式之一：

1. 是一个可执行程序，就像我们所看到的位于目录/usr/bin 中的文件一样。 属于这一类的程序，可以编译成二进制文件，诸如用 C 和 C++语言写成的程序, 也可以是由脚本语言写成的程序，比如说 shell，perl，python，ruby，等等。
2. 是一个内建于 shell 自身的命令。bash 支持若干命令，内部叫做 shell 内部命令 (builtins)。例如，cd 命令，就是一个 shell 内部命令。
3. 是一个 shell 函数。这些是小规模的 shell 脚本，它们混合到环境变量中。 在后续的章节里，我们将讨论配置环境变量以及书写 shell 函数。但是现在， 仅仅意识到它们的存在就可以了。
4. 是一个命令别名。我们可以定义自己的命令，建立在其它命令之上。

识别命令

这经常很有用，能确切地知道正在使用四类命令中的哪一类。Linux 提供了一对方法来 弄明白命令类型。

type－显示命令的类型

type 命令是 shell 内部命令，它会显示命令的类别，给出一个特定的命令名（做为参数）。 它像这样工作：

```
type command
```

"command"是你要检测的命令名。这里有些例子：

```
[me@linuxbox ~]$ type type
type is a shell builtins
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty`
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

我们看到这三个不同命令的检测结果。注意，ls 命令（在 Fedora 系统中）的检查结果，ls 命令实际上 是 ls 命令加上选项"--color=tty"的别名。现在我们知道为什么 ls 的输出结果是有颜色的！

which － 显示一个可执行程序的位置

有时候在一个操作系统中，不只安装了可执行程序的一个版本。然而在桌面系统中，这并不普遍， 但在大型服务器中，却很平常。为了确定所给定的执行程序的准确位置，使用 which 命令：

```
[me@linuxbox ~]$ which ls
/bin/ls
```

这个命令只对可执行程序有效，不包括内部命令和命令别名， 别名是真正的可执行程序的替代物。 当我们试着使用 shell 内部命令时，例如，cd 命令，我们或者得不到回应，或者是个错误信息：

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in
(/opt/jre1.6.0_03/bin:/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/opt/jre1
.6.0_03/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/home/me/bin)
```

说“命令没有找到”，真是很奇特。

得到命令文档

知道了什么是命令，现在我们来寻找每一类命令的可得到的文档。

help － 得到 shell 内部命令的帮助文档

bash 有一个内建的帮助工具，可供每一个 shell 内部命令使用。输入"help"，接着是 shell 内部命令名。例如：

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change ...
```

注意表示法：出现在命令语法说明中的方括号，表示可选的项目。一个竖杠字符 表示互斥选项。在上面 cd 命令的例子中：

```
cd [-L|-P] [dir]
```

这种表示法说明，cd 命令可能有一个"-L"选项或者"-P"选项，进一步，可能有参数"dir"。

虽然 cd 命令的帮助文档很简洁准确，但它决不是教材。正如我们所看到的，它似乎提到了许多 我们还没有谈论到的东西！不要担心，我们会学到的。

--help - 显示用法信息

许多可执行程序支持一个"--help"选项，这个选项是显示命令所支持的语法和选项说明。例如：

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create ...
```

一些程序不支持"--help"选项，但不管怎样试一下。这经常会导致输出错误信息，但同时能 揭示一样的命令用法信息。

man －显示程序手册页

许多希望被命令行使用的可执行程序，提供了一个正式的文档，叫做手册或手册页(man page)。一个特殊的叫做 man 的分页程序，可用来浏览他们。它是这样使用的：

```
man program
```

"program"是要浏览的命令名。

手册文档的格式有点不同，一般地包含一个标题，命令语法的纲要，命令用途的说明， 和命令选项列表，及每个选项的说明。然而，通常手册文档并不包含实例，它打算 作为一本参考手册，而不是教材。作为一个例子，浏览一下 ls 命令的手册文档：

```
[me@linuxbox ~]$ man ls
```

在大多数 Linux 系统中，man 使用 less 工具来显示参考手册，所以当浏览文档时，你所熟悉的 less 命令都能有效。

man 所显示的参考手册，被分成几个章节，它们不仅仅包括用户命令，也包括系统管理员 命令，程序接口，文件格式等等。下表描绘了手册的布局：

表6-1: 手册页的组织形式

章节	内容
1	用户命令
2	程序接口内核系统调用
3	C 库函数程序接口
4	特殊文件，比如说设备结点和驱动程序
5	文件格式
6	游戏娱乐，如屏幕保护程序
7	其他方面
8	系统管理员命令

有时候，我们需要查看参考手册的特定章节，从而找到我们需要的信息。 如果我们要查找一种文件格式，而同时它也是一个命令名时,这种情况尤其正确。 没有指定章节号，我们总是得到第一个匹配项，可能在第一章节。我们这样使用 man 命令，来指定章节号：

```
man section search_term
```

例如：

```
[me@linuxbox ~]$ man 5 passwd
```

命令运行结果会显示文件 /etc/passwd 的文件格式说明手册。

apropos — 显示适当的命令

也有可能搜索参考手册列表，基于某个关键字的匹配项。虽然很粗糙但有时很有用。 下面是一个以"floppy"为关键词来搜索参考手册的例子：

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8)   - udev callout to create all possible
...
```

输出结果每行的第一个字段是手册页的名字，第二个字段展示章节。注意，man 命令加上"-k"选项， 和 apropos 完成一样的

功能。

whatis 一显示非常简洁的命令说明

whatis 程序显示匹配特定关键字的手册页的名字和一行命令说明：

> 最晦涩难懂的手册页
>
> 正如我们所看到的，Linux 和类似于 Unix 的系统提供的手册页，只是打算作为参考手册使用， 而不是教材。许多手册页都很难阅读，但是我认为由于阅读难度而能拿到特等奖的手册页应该是 bash 手册页。因为我正在为这本书做我的研究，所以我很仔细地浏览了整个 bash 手册，为的是确保我讲述了 大部分的 bash 主题。当把 bash 参考手册整个打印出来，其篇幅有八十多页且内容极其紧密， 但对于初学者来说，其结构安排毫无意义。
>
> 另一方面，bash 参考手册的内容非常简明精确，同时也非常完善。所以， 如果你有胆量就查看一下， 并且期望有一天你能读懂它。

info - 显示程序 Info 条目

GNU 项目提供了一个命令程序手册页的替代物，称为 "info"。info 内容可通过 info 阅读器 程序读取。info 页是超级链接形式的，和网页很相似。这有个例子：

```
File: coreutils.info,    Node: ls invocation,    Next: dir invocation,
 Up: Directory listing

10.1 `ls': List directory contents
==================================
...
```

info 程序读取 info 文件，info 文件是树型结构，分化为各个结点，每一个包含一个题目。 info 文件包含超级链接，它可以让你从一个结点跳到另一个结点。一个超级链接可通过 它开头的星号来辨别出来，把光标放在它上面并按下 enter 键，就可以激活它。

输入"info"，接着输入程序名称， 启动 info。下表中的命令，当显示一个 info 页面时， 用来控制阅读器。

表 6－2：info 命令

命令	行为
?	显示命令帮助
PgUp or Backspace	显示上一页
PgDn or Space	显示下一页
n	下一个 - 显示下一个结点
p	上一个 - 显示上一个结点
u	Up - 显示当前所显示结点的父结点，通常是个菜单
Enter	激活光标位置下的超级链接
q	退出

到目前为止，我们所讨论的大多数命令行程序，属于 GNU 项目"coreutils"包，所以输入：

```
[me@linuxbox ~]$ info coreutils
```

将会显示一个包含超级链接的手册页，这些超级链接指向包含在 coreutils 包中的各个程序。

README 和其它程序文档

许多安装在你系统中的软件，都有自己的文档文件，这些文件位于/usr/share/doc 目录下。 这些文件大多数是以文本文件的形式存储的，可用 less 阅读器来浏览。一些文件是 HTML 格式， 可用网页浏览器来阅读。我们可能遇到许多以".gz"结尾的文件。这表示 gzip 压缩程序 已经压缩了这些程序。gzip 软件包包括一个特殊的 less 版本，叫做 zless，zless 可以显示由 gzip 压缩的文本文件的内容。

用别名（alias）创建你自己的命令

现在是时候，感受第一次编程经历了！我们将用 alias 命令创建我们自己的命令。但在 开始之前，我们需要展示一个命令行小技巧。可以把多个命令放在同一行上，命令之间 用";"分开。它像这样工作：

```
command1; command2; command3...
```

我们会用到下面的例子：

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games    kerberos  lib64    local  share  tmp
...
[me@linuxbox ~]$
```

正如我们看到的，我们在一行上联合了三个命令。首先更改目录到/usr，然后列出目录 内容，最后回到原始目录（用命令"cd ~"）,结束在开始的地方。现在，通过 alia 命令 把这一串命令转变为一个命令。我们要做的第一件事就是为我们的新命令构想一个名字。 比方说"test"。在使用"test"之前， 查明是否"test"命令名已经存在系统中，是个很不错 的主意。为了查清此事，可以使用 type 命令：

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

哦！"test"名字已经被使用了。试一下"foo":

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

太棒了！"foo"还没被占用。创建命令别名：

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

注意命令结构：

```
alias name='string'
```

在命令"alias"之后，输入“name”，紧接着（没有空格）是一个等号，等号之后是 一串用引号引起的字符串，字符串的内容要赋值给 name。我们定义了别名之后， 这个命令别名可以使用在任何地方。试一下：

```
[me@linuxbox ~]$ foo
bin   games   kerberos  lib64    local   share  tmp
...
[me@linuxbox ~]$
```

我们也可以使用 type 命令来查看我们的别名：

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -'
```

删除别名，使用 unalias 命令，像这样：

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

虽然我们有意避免使用已经存在的命令名来命名我们的别名，但这是常做的事情。通常， 会把一个普遍用到的选项加到一个经常使用的命令后面。例如，之前见到的 ls 命令，会 带有色彩支持：

```
[me@linuxbox ~]$ type ls
ls is aliased to 'ls --color=tty'
```

要查看所有定义在系统环境中的别名，使用不带参数的 alias 命令。下面在 Fedora 系统中 默认定义的别名。试着弄明白，它们是做什么的：

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
...
```

在命令行中定义 别名有点儿小问题。当你的 shell 会话结束时，它们会消失。随后的章节里， 我们会了解怎样把自己的别名添加到文件中去，每次我们登录系统，这些文件会建立系统环境。 现在，好好享受我们刚经历过的，步入 shell 编程世界的第一步吧， 虽然微小。

拜访老朋友

既然我们已经学习了怎样找到命令的帮助文档，那就试着查阅，到目前为止，我们学到的所有 命令的文档。学习命令其它可用的选项，练习一下！

拓展阅读

- 在网上，有许多关于 Linux 和命令行的文档。以下是一些最好的文档：
- Bash 参考手册是一本 bash shell 的参考指南。它仍然是一本参考书，但是包含了很多 实例，而且它比 bash 手册页容易阅读。

  http://www.gnu.org/software/bash/manual/bashref.html

- Bash FAQ 包含关于 bash，而经常提到的问题的答案。这个列表面向 bash 的中高级用户， 但它包含了许多有帮助的信息。

  http://mywiki.wooledge.org/BashFAQ

- GUN 项目为它的程序提供了大量的文档，这些文档组成了 Linux 命令行实验的核心。 这里你可以看到一个完整的列表：

  http://www.gnu.org/manual/manual.html

- Wikipedia 有一篇关于手册页的有趣文章：

  http://en.wikipedia.org/wiki/Man_page

重定向

这堂课，我们来介绍可能是命令行最酷的特性。它叫做 I/O 重定向。"I/O"代表输入/输出， 通过这个工具，你可以重定向命令的输入输出，命令的输入来自文件，而输出也存到文件。 也可以把多个命令连接起来组成一个强大的命令管道。为了炫耀这个工具，我们将叙述 以下命令：

- cat － 连接文件
- sort － 排序文本行
- uniq － 报道或省略重复行
- grep － 打印匹配行
- wc － 打印文件中换行符，字，和字节个数
- head － 输出文件第一部分
- tail - 输出文件最后一部分

标准输入，输出，和错误

到目前为止，我们用到的许多程序都会产生某种输出。这种输出，经常由两种类型组成。 第一，程序运行结果；这是说，程序要完成的功能。第二，我们得到状态和错误信息， 这些告诉我们程序进展。如果我们观察一个命令，像 ls，会看到它的运行结果和错误信息 显示在屏幕上。

与 Unix 主题“任何东西都是一个文件”保持一致，程序，比方说 ls，实际上把他们的运行结果 输送到一个叫做标准输出的特殊文件（经常用 stdout 表示），而它们的状态信息则送到另一个 叫做标准错误的文件（stderr）。默认情况下，标准输出和标准错误都连接到屏幕，而不是 保存到磁盘文件。除此之外，许多程序从一个叫做标准输入（stdin）的设备得到输入，默认情况下， 标准输入连接到键盘。

I/O 重定向允许我们可以更改输出走向和输入来向。一般地，输出送到屏幕，输入来自键盘， 但是通过 I/O 重定向，我们可以改变输入输出方向。

重定向标准输出

I/O 重定向允许我们来重定义标准输出送到哪里。重定向标准输出到另一个文件除了屏幕，我们使用 ">" 重定向符，其后跟着文件名。为什么我们要这样做呢？因为有时候把一个命令的运行结果存储到 一个文件很有用处。例如，我们可以告诉 shell 把 ls 命令的运行结果输送到文件 ls-output.txt 中去， 由文件代替屏幕。

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

这里，我们创建了一个长长的目录/usr/bin 列表，并且输送程序运行结果到文件 ls-output.txt 中。 我们检查一下重定向的命令输出结果：

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1   me   me    167878 2008-02-01 15:07 ls-output.txt
```

好；一个不错的大型文本文件。如果我们用 less 阅读器来查看这个文件，我们会看到文件 ls-output.txt 的确包含 ls 命令的执行结果。

```
[me@linuxbox ~]$ less ls-output.txt
```

现在，重复我们的重定向测试，但这次有改动。我们把目录换成一个不存在的目录。

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

我们收到一个错误信息。这很有意义，因为我们指定了一个不存在的目录/bin/usr, 但是为什么这条错误信息显示在屏幕上而不是被重定向到文件 ls-output.txt？答案是， ls 程序不把它的错误信息输送到标准输出。反而，像许多写得不错的 Unix 程序，ls 把 错误信息送到标准错误。因为我们只是重定向了标准输出，而没有重定向标准错误， 所以错误信息被送到屏幕。马上，我们将知道怎样重定向标准错误，但是首先看一下 我们的输出文件发生了什么事情。

```
me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me    0 2008-02-01 15:08 ls-output.txt
```

文件长度成为零！这是因为，当我们使用 ">" 重定向符来重定向输出结果时，目标文件总是从开头被重写。 因为我们 ls 命令没有产生运行结果，只有错误信息，重定向操作开始重写文件，然后 由于错误而停止，导致文件内容删除。事实上，如果我们需要删除一个文件内容（或者创建一个 新的空文件），可以使用这样的技巧：

```
[me@linuxbox ~]$ > ls-output.txt
```

简单地使用重定向符，没有命令在它之前，这会删除一个已存在文件的内容或是 创建一个新的空文件。

所以，怎样才能把重定向结果追加到文件内容后面，而不是从开头重写文件？为了这个目的， 我们使用">>"重定向符，像这样：

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

使用">>"操作符，将导致输出结果添加到文件内容之后。如果文件不存在，文件会 被创建，就如使用了'>'操作符。把它放到测试中：

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me    503634 2008-02-01 15:45 ls-output.txt
```

我们重复执行命令三次，导致输出文件大小是原来的三倍。

重定向标准错误

重定向标准错误缺乏专用的重定向操作符。重定向标准错误，我们必须参考它的文件描述符。 一个程序可以在几个编号的文件流中的任一个上产生输出。然而我们必须把这些文件流的前 三个看作标准输入，输出和错误，shell 内部参考它们为文件描述符0，1和2，各自地。shell 提供 了一种表示法来重定向文件，使用文件描述符。因为标准错误和文件描述符2一样，我们用这种 表示法来重定向标准错误：

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

文件描述符"2"，紧挨着放在重定向操作符之前，来执行重定向标准错误到文件 ls-error.txt 任务。

重定向标准输出和错误到同一个文件

可能有这种情况，我们希望捕捉一个命令的所有输出到一个文件。为了完成这个，我们 必须同时重定向标准输出和标准错误。有两种方法来完成任务。第一个，传统的方法， 在旧版本 shell 中也有效：

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

使用这种方法，我们完成两个重定向。首先重定向标准输出到文件 ls-output.txt，然后 重定向文件描述符2（标准错误）到文件描述符1（标准输出）使用表示法2>&1。

注意重定向的顺序安排非常重要。标准错误的重定向必须总是出现在标准输出 重定向之后，要不然它不起作用。上面的例子，

```
>ls-output.txt 2>&1
```

重定向标准错误到文件 ls-output.txt，但是如果命令顺序改为：

```
2>&1 >ls-output.txt
```

则标准错误定向到屏幕。

现在的 bash 版本提供了第二种方法，更精简合理的方法来执行这种联合的重定向。

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

在这个例子里面，我们使用单单一个表示法 &> 来重定向标准输出和错误到文件 ls-output.txt。

处理不需要的输出

有时候"沉默是金"，我们不想要一个命令的输出结果，只想把它们扔掉。这种情况 尤其适用于错误和状态信息。系统为我们提供了解决问题的方法，通过重定向输出结果 到一个特殊的叫做"/dev/null"的文件。这个文件是系统设备，叫做位存储桶，它可以 接受输入，并且对输入不做任何处理。为了隐瞒命令错误信息，我们这样做：

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

> *Unix* 文化中的*/dev/null*
>
> 位存储桶是个古老的 Unix 概念，由于它的普遍性，它的身影出现在 Unix 文化的 许多部分。当有人说他/她正在发送你的评论到/dev/null，现在你应该知道那是 什么意思了。更多的例子，可以阅读 Wikipedia 关于"/dev/null"的文章。

重定向标准输入

到目前为止，我们还没有遇到一个命令是利用标准输入的（实际上我们遇到过了，但是 一会儿再揭晓谜底），所以我们需要介绍一个。

cat － 连接文件

cat 命令读取一个或多个文件，然后复制它们到标准输出，就像这样:

```
cat [file]
```

在大多数情况下，你可以认为 cat 命令相似于 DOS 中的 TYPE 命令。你可以使用 cat 来显示 文件而没有分页，例如：

```
[me@linuxbox ~]$ cat ls-output.txt
```

将会显示文件 ls-output.txt 的内容。cat 经常被用来显示简短的文本文件。因为 cat 可以 接受不只一个文件作为参数，所以它也可以用来把文件连接在一起。比方说我们下载了一个 大型文件，这个文件被分离成多个部分（USENET 中的多媒体文件经常以这种方式分离）， 我们想把它们连起来。如果文件命名为：

我们能用这个命令把它们连接起来：

```
cat movie.mpeg.0* > movie.mpeg
```

因为通配符总是以有序的方式展开，所以这些参数会以正确顺序安排。

这很好，但是这和标准输入有什么关系呢？没有任何关系，让我们试着做些其他的工作。 如果我们输入不带参数的"cat"命令，会发生什么呢：

```
[me@linuxbox ~]$ cat
```

没有发生任何事情，它只是坐在那里，好像挂掉了一样。看起来是那样，但是它正在做它该做的事情：

如果 cat 没有给出任何参数，它会从标准输入读入数据，因为标准输入，默认情况下，连接到键盘。 它正在等待我们输入数据！试试这个：

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

下一步，输入 Ctrl-d（按住 Ctrl 键同时按下"d"），来告诉 cat，在标准输入中， 它已经到达文件末尾（EOF）：

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

由于文件名参数的缺席，cat 复制标准输入到标准输出，所以我们看到文本行重复出现。 我们可以使用这种行为来创建简短的文本文件。比方说，我们想创建一个叫做"lazy_dog.txt" 的文件，这个文件包含例子中的文本。我们这样做：

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

输入命令，其后输入要放入文件中的文本。记住，最后输入 Ctrl-d。通过使用这个命令，我们 实现了世界上最低能的文字处理器！看一下运行结果，我们使用 cat 来复制文件内容到 标准输出：

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

现在我们知道怎讲接受标准输入，除了文件名参数，让我们试着重定向标准输入：

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

使用“<”重定向操作符，我们把标准输入源从键盘改到文件 lazy_dog.tx。我们看到结果 和传递单个文件名作为参数的执行结果一样。把这和传递一个文件名参数作比较，尤其没有意义， 但它是用来说明把一个文件作为标准输入源。

在我们继续之前，查看 cat 的手册页，因为它有几个有趣的选项。

管道线

命令可以从标准输入读取数据，然后再把数据输送到标准输出，命令的这种能力被 一个 shell 特性所利用，这个特性叫做管道线。使用管道操作符"|"（竖杠），一个命令的 标准输出可以管道到另一个命令的标准输入：

```
command1 | command2
```

为了全面地说明这个命令，我们需要一些命令。是否记得我们说过，我们已经知道有一个 命令接受标准输入？它是 less 命令。我们用 less 来一页一页地显示任何命令的输出，命令把 它的运行结果输送到标准输出：

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

这极其方便！使用这项技术，我们可以方便地检测会产生标准输出的任一命令的运行结果。

过滤器

管道线经常用来对数据完成复杂的操作。有可能会把几个命令放在一起组成一个管道线。 通常，以这种方式使用的命令被称为过滤器。过滤器接受输入，以某种方式改变它，然后 输出它。第一个我们想试验的过滤器是 sort。想象一下，我们想把目录/bin 和/usr/bin 中 的可执行程序都联合在一起，再把它们排序，然后浏览执行结果：

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

因为我们指定了两个目录（/bin 和/usr/bin），ls 命令的输出结果由有序列表组成， 各自针对一个目录。通过在管道线中包含 sort，我们改变输出数据，从而产生一个 有序列表。

uniq - 报道或忽略重复行

uniq 命令经常和 sort 命令结合在一起使用。uniq 从标准输入或单个文件名参数接受数据有序 列表（详情查看 uniq 手册页），默认情况下，从数据列表中删除任何重复行。所以，为了确信 我们的列表中不包含重复句子（这是说，出现在目录/bin 和/usr/bin 中重名的程序），我们添加 uniq 到我们的管道线中：

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

在这个例子中，我们使用 uniq 从 sort 命令的输出结果中，来删除任何重复行。如果我们想看到 重复的数据列表，让 uniq 命令带上"-d"选项，就像这样：

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc 一打印行，字和字节数

wc（字计数）命令是用来显示文件所包含的行，字和字节数。例如：

```
[me@linuxbox ~]$ wc ls-output.txt
7902 64566 503634 ls-output.txt
```

在这个例子中，wc 打印出来三个数字：包含在文件 ls-output.txt 中的行数，单词数和字节数， 正如我们先前的命令，如果 wc 不带命令行参数，它接受标准输入。"-l"选项限制命令输出只能 报道行数。添加 wc 到管道线来统计数据，是个很便利的方法。查看我们的有序列表中程序个数， 我们可以这样做：

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep －打印匹配行

grep 是个很强大的程序，用来找到文件中的匹配文本。这样使用 grep 命令：

```
grep pattern [file...]
```

当 grep 遇到一个文件中的匹配"模式"，它会打印出包含这个类型的行。grep 能够匹配的模式可以 很复杂，但是现在我们把注意力集中在简单文本匹配上面。在后面的章节中，我们将会研究 高级模式，叫做正则表达式。

比如说，我们想在我们的程序列表中，找到文件名中包含单词"zip"的所有文件。这样一个搜索， 可能让我们了解系统中的一些程序与文件压缩有关系。这样做：

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
...
```

grep 有一对方便的选项："-i"导致 grep 忽略大小写当执行搜索时（通常，搜索是大小写 敏感的），"-v"选项会告诉 grep 只打印不匹配的行。

head / tail －打印文件开头部分/结尾部分

有时候你不需要一个命令的所有输出。可能你只想要前几行或者后几行的输出内容。 head 命令打印文件的前十行，而 tail 命令打印文件的后十行。默认情况下， 两个命令 都打印十行文本，但是可以通过"-n"选项来调整命令打印的行数。

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
...
[me@linuxbox ~]$ tail -n 5 ls-output.txt
...
```

它们也能用在管道线中：

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
...
```

tail 有一个选项允许你实时的浏览文件。当观察日志文件的进展时，这很有用，因为 它们同时在被写入。在以下的例子里，我们要查看目录/var/log 里面的信息文件。在 一些 Linux 发行版中，要求有超级用户权限才能阅读这些文件，因为文件/var/log/messages 可能包含安全信息。

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb 8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
....
```

使用"-f"选项，tail 命令继续监测这个文件，当新的内容添加到文件后，它们会立即 出现在屏幕上。这会一直继续下去直到你输入 Ctrl-c。

tee － 从 Stdin 读取数据，并同时输出到 Stdout 和文件

为了和我们的管道隐喻保持一致，Linux 提供了一个叫做 tee 的命令，这个命令制造了 一个"tee"，安装到我们的管道上。tee 程序从标准输入读入数据，并且同时复制数据 到标准输出（允许数据继续随着管道线流动）和一个或多个文件。当在某个中间处理 阶段来捕捉一个管道线的内容时，这很有帮助。这里，我们重复执行一个先前的例子， 这次包含 tee 命令，在 grep 过滤管道线的内容之前，来捕捉整个目录列表到文件 ls.txt：

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
....
```

总结归纳

一如既往，查看这章学到的每一个命令的文档。我们已经知道了他们最基本的用法。 它们还有很多有趣的选项。随着我们 Linux 经验的积累，我们会了解命令行重定向特性 在解决特殊问题时非常有用处。有许多命令利用标准输入和输出，而几乎所有的命令行 程序都使用标准错误来显示它们的详细信息。

> *Linux* 可以激发我们的想象
>
> 当我被要求解释 Windows 与 Linux 之间的差异时，我经常拿玩具来作比喻。
>
> Windows 就像一个游戏机。你去商店，买了一个包装在盒子里面的全新的游戏机。 你把它带回家，打开盒子，开始玩游戏。精美的画面，动人的声音。玩了一段时间之后， 你厌倦了它自带的游戏，所以你返回商店，又买了另一个游戏机。这个过程反复重复。 最后，你玩腻了游戏机自带的游戏，你回到商店，告诉售货员，“我想要一个这样的游戏！” 但售货员告诉你没有这样的游戏存在，因为它没有“市场需求”。然后你说，“但是我只 需要修改一下这个游戏！“，售货员又告诉你不能修改它。所有游戏都被封装在它们的 存储器中。到头来，你发现你的玩具只局限于别人为你规定好的游戏。
>
> 另一方面，Linux 就像一个全世界上最大的建造模型。你打开它， 发现它只是一个巨大的 部件集合。有许多钢支柱，螺钉，螺母，齿轮，滑轮，发动机，和一些怎样来建造它的说明书。 然后你开始摆弄它。你建造了一个又一个样板模型。过了一会儿，你发现你要建造自己的模型。 你不必返回商店，因为你已经拥有了你需要的一切。建造模型以你构想的形状为模板，搭建 你想要的模型。
>
> 当然，选择哪一个玩具，是你的事情，那么你觉得哪个玩具更令人满意呢？

从 shell 眼中看世界

在这一章我们将看一下，当你按下 enter 键后，发生在命令行中的一些“魔法”。虽然我们会 仔细查看几个复杂有趣的 shell 特点，但我们只使用一个新命令来处理这些特性。

echo — 显示一行文本

(字符)展开

每一次你输入一个命令，然后按下 enter 键，在 bash 执行你的命令之前，bash 会对输入 的字符完成几个步骤处理。我们已经知道两三个案例，怎样一个简单的字符序列，例如"*", 对 shell 来说，有很多的涵义。使这个发生的过程叫做（字符）展开。通过展开， 你输入的字符，在 shell 对它起作用之前，会展开成为别的字符。为了说明我们所要 表达的意思，让我们看一看 echo 命令。echo 是一个 shell 内部命令，来完成非常简单的认为。 它在标准输出中打印出它的文本参数。

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

这个命令的作用相当简单明了。传递到 echo 命令的任一个参数都会在（屏幕上）显示出来。 让我们试试另一个例子：

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

那么刚才发生了什么事情呢？ 为什么 echo 不打印"*"呢？随着你回想起我们所学过的 关于通配符的内容，这个"*"字符意味着匹配文件名中的任意字符，但是在原先的讨论 中我们却不知道 shell 是怎样实现这个功能的。最简单的答案就是shell把"*"展开成了 另外的东西（在这种情况下，就是在当前工作目录下的文件名字），在echo命令被执行 前。当回车键被按下时，shell在命令被执行前在命令行上自动展开任何符合条件的字符， 所以echo命令从不会发现"*",只把它展开成结果。知道了这个以后，我们能看到echo执行 的结果和我们想象的一样。

路径名展开

这种通配符工作机制叫做路径名展开。如果我们试一下在之前的章节中使用的技巧， 我们会看到它们真是要展开的字符。给出一个主目录，它看起来像这样：

```
[me@linuxbox ~]$ ls
Desktop   ls-output.txt   Pictures   Templates
....
```

我们能够执行以下参数展开模式：

```
[me@linuxbox ~]$ echo D*
Desktop  Documents
```

和：

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

甚至是：

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

查看主目录之外的目录：

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share  /usr/local/share
```

> 隐藏文件路径名展开
>
> 正如我们知道的，以圆点字符开头的文件名是隐藏文件。路径名展开也尊重这种 行为。像这样的展开：
>
> echo *
>
> 不会显示隐藏文件
>
> 要是展开模式以一个圆点开头，我们就能够在展开模式中包含隐藏文件， 而且隐藏文件可能会出现在第一位置，就像这样：
>
> echo .*
>
> 它几乎是起作用了。然而，如果我们仔细检查一下输出结果，我们会看到名字"." 和".."也出现在结果中。因为这些名字是指当前工作目录和它的父目录，使用这种 模式可能会产生不正确的结果。我们能看到这样的结果，如果我们试一下这个命令：
>
> ls -d .* | less
>
> 为了在这种情况下正确地完成路径名展开，我们应该雇佣一个更精确些的模式。 这个模式会正确地工作：
>
> ls -d .[!.]?*
>
> 这种模式展开成为文件名，每个文件名以圆点开头，第二个字符不包含圆点，再包含至少一个字符， 并且这个字符之后紧接着任意多个字符。这将列出大多数的隐藏文件 （但仍将不能包含以多个圆点开头的文件名）这个带有 -A 选项（“几乎所有”）的 ls 命令能够提供一份正确的隐藏文件清单：
>
> ls -A

波浪线展开

可能你从我们对 cd 命令的介绍中回想起来，波浪线字符("~")有特殊的意思。当它用在一个 单词的开头时，它会展开成指定用户的主目录名，如果没有指定用户名，则是当前用户的主目录：

```
[me@linuxbox ~]$ echo ~
/home/me
```

如果有用户"foo"这个帐号，然后：

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

算术表达式展开

shell 允许算术表达式通过展开来执行。这允许我们把 shell 提示当作计算器来使用：

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

算术表达式展开使用这种格式：

```
$((expression))
```

（以上括号中的）表达式是指算术表达式，它由数值和算术操作符组成。

算术表达式只支持整数（全部是数字，不带小数点），但是能执行很多不同的操作。这里是 一些它支持的操作符：

表 8-1: 算术操作符

操作符	说明
+	加
-	减
	乘
/	除（但是记住，因为展开只是支持整数除法，所以结果是整数。）
%	取余，只是简单的意味着，“余数”
*	取幂

在算术表达式中空格并不重要，并且表达式可以嵌套。例如，5的平方乘以3：

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

一对括号可以用来把多个子表达式括起来。通过这个技术，我们可以重写上面的例子， 同时用一个展开代替两个，来得到一样的结果：

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

这个一个使用除法和取余操作符的例子。注意整数除法的结果：

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

在35章会更深入的讨论算术表达式的内容。

花括号展开

可能最奇怪的展开是花括号展开。通过它，你可以从一个包含花括号的模式中 创建多个文本字符串。这是一个例子：

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

花括号展开模式可能包含一个开头部分叫做报头，一个结尾部分叫做附言。花括号表达式本身可 能包含一个由逗号分开的字符串列表，或者一系列整数，或者单个的字符串。这种模式可能 不包括嵌入的空白。这个例题使用了一系列整数：

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1  Number_2  Number_3  Number_4  Number_5
```

一系列以倒序排列的字母：

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

花括号展开可以嵌套：

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

那么这对什么有好处呢？最普遍的应用是，创建一系列的文件或目录列表。例如， 如果我们是摄影师，有大量的相片。我们想把这些相片按年月先后组织起来。首先， 我们要创建一系列以数值"年－月"形式命名的目录。通过这种方式，目录名按照年代顺序排列。我们可以键入整个目录列表，但是工作量太大了，并且易于出错。 反而，我们可以这样做：

```
[me@linuxbox ~]$ mkdir Pics
[me@linuxbox ~]$ cd Pics
[me@linuxbox Pics]$ mkdir {2007..2009}-0{1..9} {2007..2009}-{10..12}
[me@linuxbox Pics]$ ls
2007-01 2007-07 2008-01 2008-07 2009-01 2009-07
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

棒极了！

参数展开

在这一章我们将会简单地介绍参数展开，只是皮毛而已。后续章节我们会广泛地 讨论参数展开。这个特性在 shell 脚本中比直接在命令行中更有用。它的许多性能 和系统存储小块数据，并给每块数据命名的能力有关系。许多像这样的小块数据， 更适当些应叫做变量，可以方便地检查它们。例如，叫做"USER"的变量包含你的 用户名。唤醒参数展开，揭示 USER 中的内容，可以这样做：

```
[me@linuxbox ~]$ echo $USER
me
```

查看有效的变量列表，试试这个：

```
[me@linuxbox ~]$ printenv | less
```

你可能注意到其它展开类型，如果你误输入一个模式，展开就不会发生。这时 echo 命令只简单地显示误键入的模式。通过参数展开，如果你拼写错了一个变量名， 展开仍然会进行，只是展成一个空字符串：

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

命令替换

命令替换允许我们把一个命令的输出作为一个展开模式来使用：

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos

[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

这里我们把 which cp 的执行结果作为一个参数传递给 ls 命令，因此要想得到 cp 程序的 输出列表，不必知道它完整的路径名。我们不只限制于简单命令。也可以使用整个管道线 （只展示部分输出）：

```
[me@linuxbox ~]$ file $(ls /usr/bin/* | grep zip)
/usr/bin/bunzip2:     symbolic link to `bzip2'
....
```

在这个例子中，管道线的输出结果成为 file 命令的参数列表。

在旧版 shell 程序中，有另一种语法也支持命令替换，可与刚提到的语法轮换使用。 bash 也支持这种语法。它使用倒引号来代替美元符号和括号：

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

引用

我们已经知道 shell 有许多方式可以完成展开，现在是时候学习怎样来控制展开了。 以下面例子来说明：

```
[me@linuxbox ~]$ echo this is a    test
this is a test
```

或者：

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

在第一个例子中，shell 从 echo 命令的参数列表中，删除多余的空格。在第二个例子中， 参数展开把 `$1` 的值替换为一个空字符串，因为 `1` 是没有定义的变量。shell 提供了一种 叫做引用的机制，来有选择地禁止不需要的展开。

双引号

我们将要看一下引用的第一种类型，双引号。如果你把文本放在双引号中， shell 使用的特殊字符，除了 $，\ (反斜杠），和 `（倒引号）之外， 则失去它们的特殊含义，被当作普通字符来看待。这意味着单词分割，路径名展开， 波浪线展开，和花括号展开都被禁止，然而参数展开，算术展开，和命令替换 仍然执行。使用双引号，我们可以处理包含空格的文件名。比方说我们是不幸的 名为 *two words.txt* 文件的受害者。如果我们试图在命令行中使用这个 文件，单词分割机制会导致这个文件名被看作两个独自的参数，而不是所期望 的单个参数：

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

使用双引号，我们可以阻止单词分割，得到期望的结果；进一步，我们甚至可以修复 破损的文件名。

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me   me   18 2008-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

你瞧！现在我们不必一直输入那些讨厌的双引号了。

记住，在双引号中，参数展开，算术表达式展开，和命令替换仍然有效：

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4    February 2008
Su Mo Tu We Th Fr Sa
....
```

我们应该花费一点时间来看一下双引号在命令替换中的效果。首先仔细研究一下单词分割 是怎样工作的。在之前的范例中，我们已经看到单词分割机制是怎样来删除文本中额外空格的：

```
[me@linuxbox ~]$ echo this is a   test
this is a test
```

在默认情况下，单词分割机制会在单词中寻找空格，制表符，和换行符，并把它们看作 单词之间的界定符。它们只作为分隔符使用。因为它们把单词分为不同的参数，在范例中， 命令行包含一个带有四个不同参数的命令。如果我们加上双引号：

```
[me@linuxbox ~]$ echo "this is a    test"
this is a    test
```

单词分割被禁止，内嵌的空格也不会被当作界定符，它们成为参数的一部分。 一旦加上双引号，我们的命令行就包含一个带有一个参数的命令。

事实上，单词分割机制把换行符看作界定符，对命令替换产生了一个，虽然微妙，但有趣的影响。 考虑下面的例子：

```
[me@linuxbox ~]$ echo $(cal)
February 2008 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
February 2008
....
```

在第一个实例中，没有引用的命令替换导致命令行包含38个参数。在第二个例子中， 命令行只有一个参数，参数中包括嵌入的空格和换行符。

单引号

如果需要禁止所有的展开，我们使用单引号。以下例子是无引用，双引号，和单引号的比较结果：

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt   {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt  {a,b} $(echo foo) $((2+2)) $USER
```

正如我们所看到的，随着引用程度加强，越来越多的展开被禁止。

转义字符

有时候我们只想引用单个字符。我们可以在字符之前加上一个反斜杠，在这个上下文中叫做转义字符。 经常在双引号中使用转义字符，来有选择地阻止展开。

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

使用转义字符来消除文件名中一个字符的特殊含义，是很普遍的。例如，在文件名中可能使用 一些对于 shell 来说，有特殊含义的字符。这些字符包括"$", "!", " "等字符。在文件名 中包含特殊字符，你可以这样做：

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

为了允许反斜杠字符出现，输入"\\"来转义。注意在单引号中，反斜杠失去它的特殊含义，它 被看作普通字符。

> 反斜杠转义字符序列
>
> 反斜杠除了作为转义字符外，反斜杠也是一种表示法的一部分，这种表示法代表某种 特殊字符，叫做控制码。ASCII 编码表中前32个字符被用来把命令转输到像电报机 一样的设备。一些编码是众所周知的（制表符，退格符，换行符，和回车符），其它 一些编码就不熟悉了（空值，传输结束码，和确认）。
>
> | 转义序列 | 意思 |
> |---|---|
> | \a | 响铃（"警告"－导致计算机嘟嘟响 |
> | \b | 退格符 |
> | \n | 新的一行。在类似 Unix 系统中，产生换行。 |
> | \r | 回车符 |
> | \t | 制表符 |
>
> 上表列出了一些常见的反斜杠转义字符。反斜杠表示法背后的思想来源于 C 编程语言， 许多其它语言也采用了这种表示方法，包括 shell。
>
> echo 命令带上"-e"选项，能够解释转义序列。你可以把转义序列放在$' '里面。 以下例子，使用 sleep 命令，一个简单的程序，它会等待指定的秒数，然后退出。 我们可以创建一个简单的倒数计数器：
>
> *sleep 10; echo -e "Time's up\a"*
>
> 我们也可以这样做：
>
> *sleep 10; echo "Time's up" $'\a'*

总结归纳

随着我们继续学习 shell，你会发现使用展开和引用的频率逐渐多起来，所以能够很好的 理解他们的工作方式很有意义。事实上，可以这样说，他们是学习 shell 的最重要的主题。 如果没有准确地理解展开模式，shell 总是神秘和混乱的源泉，并且 shell 潜在的能力也 浪费掉了。

拓展阅读

- Bash 手册页有主要段落是关于展开和引用的，它们以更正式的方式介绍了这些题目。
- Bash 参考手册也包含章节，介绍展开和引用：

  http://www.gnu.org/software/bash/manual/bashref.html

键盘高级操作技巧

开玩笑地说，我经常把 Unix 描述为“这个操作系统是为喜欢敲键盘的人们服务的。” 当然，Unix 甚至还有一个命令行，这个事实是个确凿的证据，证明了我所说的话。 但是命令行用户不喜欢敲入那么多字。那又为什么如此多的命令会有这样简短的命令名，像 cp，ls，mv，和 rm？事实上，命令行最为珍视的目标之一就是懒惰；用最少的击键次数 来完成最多的工作。另一个目标是你的手指永远不必离开键盘，永不触摸鼠标。在这一 章节，我们将看一下 bash 特性，这些特性使键盘使用起来更加迅速，更加高效。

以下命令将会露面：

- clear － 清空屏幕
- history － 显示历史列表内容

命令行编辑

Bash 使用了一个名为 Readline 的库（共享的线程集合，可以被不同的程序使用）， 来实现命令行编辑。我们已经看到一些例子。我们知道，例如，箭头按键可以移动鼠标， 此外还有许多特性。想想这些额外的工具，我们可以在工作中使用。学会所有的特性 并不重要，但许多特性非常有帮助。选择自己需要的特性。

注意：下面一些按键组合（尤其使用 Alt 键的组合）， 可能会被 GUI 拦截来触发其它的功能。 当使用虚拟控制台时，所有的按键组合都应该正确地工作。

移动光标

下表列出了移动光标所使用的按键：

表9-1: 光标移动命令

按键	行动
Ctrl-a	移动光标到行首。
Ctrl-e	移动光标到行尾。
Ctrl-f	光标前移一个字符；和右箭头作用一样。
Ctrl-b	光标后移一个字符；和左箭头作用一样。
Alt-f	光标前移一个字。
Alt-b	光标后移一个字。
Ctrl-l	清空屏幕，移动光标到左上角。clear 命令完成同样的工作。

修改文本

表9－2列出了键盘命令，这些命令用来在命令行中编辑字符。

表9-2: 文本编辑命令

按键	行动
Ctrl-d	删除光标位置的字符。
Ctrl-t	光标位置的字符和光标前面的字符互换位置。
Alt-t	光标位置的字和其前面的字互换位置。
Alt-l	把从光标位置到字尾的字符转换成小写字母。
Alt-u	把从光标位置到字尾的字符转换成大写字母。

剪切和粘贴文本

Readline 的文档使用术语 killing 和 yanking 来指我们平常所说的剪切和粘贴。 剪切下来的本文被存储在一个叫做剪切环(kill-ring)的缓冲区中。

表9-3: 剪切和粘贴命令

按键	行动
Ctrl-k	剪切从光标位置到行尾的文本。
Ctrl-u	剪切从光标位置到行首的文本。
Alt-d	剪切从光标位置到词尾的文本。
Alt-Backspace	剪切从光标位置到词头的文本。如果光标在一个单词的开头，剪切前一个单词。
Ctrl-y	把剪切环中的文本粘贴到光标位置。

> 元键
>
> 如果你冒险进入到 Readline 的文档中，你会在 bash 手册页的 READLINE 段落， 遇到一个术语"元键"（meta key）。在当今的键盘上，这个元键是指 Alt 键，但 并不总是这样。
>
> 回到昏暗的年代（在 PC 之前 Unix 之后）， 并不是每个人都有他们自己的计算机。 他们可能有一个叫做终端的设备。 一个终端是一种通信设备，它以一个文本显示 屏幕和一个键盘作为其特色，它里面有足够的电子器件来显示文本字符和移动光标。 它连接到（通常通过串行电缆）一个更大的计算机或者是一个大型计算机的通信 网络。有许多不同的终端产品商标，它们有着不同的键盘和特征显示集。因为它们 都倾向于至少能理解 ASCII，所以软件开发者想要符合最低标准的可移植的应用程序。 Unix 系统有一个非常精巧的方法来处理各种终端产品和它们不同的显示特征。因为 Readline 程序的开发者们，不能确定一个专用多余的控制键的存在，他们发明了一个 控制键，并把它叫做"元"（"meta"）。然而在现代的键盘上，Alt 键作为元键来服务。 如果你仍然在使用终端（在 Linux 中，你仍然可以得到一个终端），你也可以按下和 释放 Esc 键来得到如控制 Alt 键一样的效果。

自动补全

shell 能帮助你的另一种方式是通过一种叫做自动补全的机制。当你敲入一个命令时， 按下 tab 键，自动补全就会发生。让我们看一下这是怎样工作的。给出一个看起来 像这样的主目录：

```
[me@linuxbox ~]$ ls
Desktop   ls-output.txt   Pictures   Templates   Videos
....
```

试着输入下面的命令，但不要按下 Enter 键：

```
[me@linuxbox ~]$ ls l
```

现在按下 tab 键：

```
[me@linuxbox ~]$ ls ls-output.txt
```

看一下 shell 是怎样补全这一行的？让我们再试试另一个例子。这回，也 不要按下 Enter:

```
[me@linuxbox ~]$ ls D
```

按下 tab:

```
[me@linuxbox ~]$ ls D
```

没有补全，只是嘟嘟响。因为"D"不止匹配目录中的一个条目。为了自动补全执行成功， 你给它的"线索"必须不模棱两可。如果我们继续输入：

```
[me@linuxbox ~]$ ls Do
```

然后按下 tab：

```
[me@linuxbox ~]$ ls Documents
```

自动补全成功了。

这个实例展示了路径名自动补全，这是最常用的形式。自动补全也能对变量起作用（如果 字的开头是一个"$"），用户名字（单词以"~"开始），命令（如果单词是一行的第一个单词）， 和主机名（如果单词的开头是"@"）。主机名自动补全只对包含在文件/etc/hosts 中的主机名有效。

有一系列的控制和元键序列与自动补全相关联：

表9-4: 自动补全命令

按键	行动
Alt-?	显示可能的自动补全列表。在大多数系统中，你也可以完成这个通过按 两次 tab 键，这会更容易些。
Alt-*	插入所有可能的自动补全。当你想要使用多个可能的匹配项时，这个很有帮助。

> 可编程自动补全
>
> 目前的 bash 版本有一个叫做可编程自动补全工具。可编程自动补全允许你（更可能是，你的 发行版提供商）来加入额外的自动补全规则。通常需要加入对特定应用程序的支持，来完成这个 任务。例如，有可能为一个命令的选项列表，或者一个应用程序支持的特殊文件类型加入自动补全。 默认情况下，Ubuntu 已经定义了一个相当大的规则集合。可编程自动补全是由 shell 函数实现的， shell 函数是一种小巧的 shell 脚本，我们会在后面的章节中讨论到。如果你感到好奇，试一下：
>
> *set | less*
>
> 查看一下如果你能找到它们的话。默认情况下，并不是所有的发行版都包括它们。

利用历史命令

正如我们在第二章中讨论到的，bash 维护着一个已经执行过的命令的历史列表。这个命令列表 被保存在你主目录下，一个叫做 .bash_history 的文件里。这个 history 工具是个有用资源， 因为它可以减少你敲键盘的次数，尤其当和命令行编辑联系起来时。

搜索历史命令

在任何时候，我们都可以浏览历史列表的内容，通过：

```
[me@linuxbox ~]$ history | less
```

在默认情况下，bash 会存储你所输入的最后 500 个命令。在随后的章节里，我们会知道 怎样调整这个数值。比方说我们想要找到列出目录 /usr/bin 内容的命令。一种方法，我们可以这样做：

```
[me@linuxbox ~]$ history | grep /usr/bin
```

比方说在我们的搜索结果之中，我们得到一行，包含了有趣的命令，像这样；

```
88  ls -l /usr/bin > ls-output.txt
```

数字 "88" 是这个命令在历史列表中的行号。随后在使用另一种展开类型时，叫做 历史命令展开，我们会用到这个数字。我们可以这样做，来使用我们所发现的行：

```
[me@linuxbox ~]$ !88
```

bash 会把 "!88" 展开成为历史列表中88行的内容。还有其它的历史命令展开形式，我们一会儿 讨论它们。bash 也具有按递增顺序来搜索历史列表的能力。这意味着随着字符的输入，我们 可以告诉 bash 去搜索历史列表，每一个附加字符都进一步提炼我们的搜索。启动递增搜索， 输入 Ctrl-r，其后输入你要寻找的文本。当你找到它以后，你可以敲入 Enter 来执行命令， 或者输入 Ctrl-j，从历史列表中复制这一行到当前命令行。再次输入 Ctrl-r，来找到下一个 匹配项（向上移动历史列表）。输入 Ctrl-g 或者 Ctrl-c，退出搜索。实际来体验一下：

```
[me@linuxbox ~]$
```

首先输入 Ctrl-r:

```
(reverse-i-search)`':
```

提示符改变，显示我们正在执行反向递增搜索。搜索过程是"反向的"，因为我们按照从"现在"到过去 某个时间段的顺序来搜寻。下一步，我们开始输入要查找的文本。在这个例子里是 "/usr/bin"：

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

即刻，搜索返回我们需要的结果。我们可以执行这个命令，按下 Enter 键，或者我们可以复制 这个命令到我们当前的命令行，来进一步编辑它，输入 Ctrl-j。复制它，输入 Ctrl-j：

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

我们的 shell 提示符重新出现，命令行加载完毕，正准备行动！下表列出了一些按键组合， 这些按键用来操作历史列表：

表9-5: 历史命令

按键	行为
Ctrl-p	移动到上一个历史条目。类似于上箭头按键。
Ctrl-n	移动到下一个历史条目。类似于下箭头按键。
Alt-<	移动到历史列表开头。
Alt->	移动到历史列表结尾，即当前命令行。
Ctrl-r	反向递增搜索。从当前命令行开始，向上递增搜索。
Alt-p	反向搜索，不是递增顺序。输入要查找的字符串，然后按下 Enter，执行搜索。
Alt-n	向前搜索，非递增顺序。
Ctrl-o	执行历史列表中的当前项，并移到下一个。如果你想要执行历史列表中一系列的命令，这很方便。

历史命令展开

通过使用 "!" 字符，shell 为历史列表中的命令，提供了一个特殊的展开类型。我们已经知道一个感叹号， 其后再加上一个数字，可以把来自历史列表中的命令插入到命令行中。还有一些其它的展开特性：

表9-6: 历史展开命令

序列	行为
!!	重复最后一次执行的命令。可能按下上箭头按键和 enter 键更容易些。
!number	重复历史列表中第 number 行的命令。
!string	重复最近历史列表中，以这个字符串开头的命令。
!?string	重复最近历史列表中，包含这个字符串的命令。

应该小心谨慎地使用 "!string" 和 "!?string" 格式，除非你完全确信历史列表条目的内容。

在历史展开机制中，还有许多可利用的特点，但是这个题目已经太晦涩难懂了， 如果我们再继续讨论的话，我们的头可能要爆炸了。bash 手册页的 HISTORY EXPANSION 部分详尽地讲述了所有要素。

> 脚本
>
> 除了 bash 中的命令历史特性，许多 Linux 发行版包括一个叫做 script 的程序， 这个程序可以记录整个 shell 会话，并把 shell 会话存在一个文件里面。这个命令的基本语法是：
>
> script [file]
>
> 命令中的 file 是指用来存储 shell 会话记录的文件名。如果没有指定文件名，则使用文件 typescript。查看脚本的手册页， 可以得到一个关于 script 程序选项和特点的完整列表。

总结归纳

在这一章中，我们已经讨论了一些由 shell 提供的键盘操作技巧， 这些技巧是来帮助打字员减少工作量的。 随着时光流逝，你和命令行打交道越来越多， 我猜想你会重新翻阅这一章的内容，学会更多的技巧。 目前，你就认为它们是可选的，潜在地有帮助的。

拓展阅读

- Wikipedia 上有一篇关于计算机终端的好文章：

  http://en.wikipedia.org/wiki/Computer_terminal

权限

Unix 传统中的操作系统不同于那些 MS-DOS 传统中的系统，区别在于它们不仅是多任务系统，而且也是 多用户系统。这到底意味着什么？它意味着多个用户可以在同一时间使用同一台计算机。然而一个 典型的计算机可能只有一个键盘和一个监视器，但是它仍然可以被多个用户使用。例如，如果一台 计算机连接到一个网络或者因特网，那么远程用户通过 ssh（安全 shell）可以登录并操纵这台电脑。 事实上，远程用户也能运行图形界面应用程序，并且图形化的输出结果会出现在远端的显示器上。 X 窗口系统把这个作为基本设计理念的一部分，并支持这种功能。

Linux 系统的多用户性能，不是最近的“创新”，而是一种特性，它深深地嵌入到了 Linux 操作系统的 设计过程中。想一下 Unix 系统的诞生环境，这会很有意义。多年前，在个人电脑出现之前，计算机 都是大型的，昂贵的，集中化的。一个典型的大学计算机系统，例如，是由坐落在一座建筑中的一台 大型中央计算机和许多散布在校园各处的终端机组成，每个终端都连接到这台大型中央计算机。 这台计算机可以同时支持很多用户。

为了使多用户特性付诸实践，那么必须发明一种方法来阻止用户彼此之间受到影响。毕竟，一个 用户的行为不能导致计算机崩溃，也不能乱动属于另一个用户的文件。

在这一章中，我们将看看这一系统安全的重要组成部分，会介绍以下命令：

- id – 显示用户身份号
- chmod – 更改文件模式
- umask – 设置默认的文件权限
- su – 以另一个用户的身份来运行 shell
- sudo – 以另一个用户的身份来执行命令
- chown – 更改文件所有者
- chgrp – 更改文件组所有权
- passwd – 更改用户密码

拥有者，组成员，和其他人

在第四章探究文件系统时，当我们试图查看一个像 /etc/shadow 那样的文件的时候，我们会遇到一个问题。

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow:  regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow:  Permission denied
```

产生这种错误信息的原因是，作为一个普通用户，我们没有权限来读取这个文件。

在 Unix 安全模型中，一个用户可能拥有文件和目录。当一个用户拥有一个文件或目录时， 用户对这个文件或目录的访问权限拥有控制权。用户，反过来，又属于一个由一个或多个 用户组成的用户组，用户组成员由文件和目录的所有者授予对文件和目录的访问权限。除了 对一个用户组授予权限之外，文件所有者可能会给每个人一些权限，在 Unix 术语中，每个人 是指整个世界。可以用 id 命令，来找到关于你自己身份的信息：

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

让我们看一下输出结果。当用户创建帐户之后，系统会给用户分配一个号码，叫做用户 ID 或者 uid，然后，为了符合人类的习惯，这个 ID 映射到一个用户名。系统又会给这个用户 分配一个原始的组 ID 或者是 gid，这个 gid 可能属于另外的组。上面的例子来自于 Fedora 系统， 比方说 Ubuntu 的输出结果可能看起来有点儿不同：

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(me)
```

正如我们能看到的，两个系统中用户的 uid 和 gid 号码是不同的。原因很简单，因为 Fedora 系统 从500开始进行普通用户帐户的编号，而 Ubuntu 从1000开始。我们也能看到 Ubuntu 的用户属于 更多的用户组。这和 Ubuntu 管理系统设备和服务权限的方式有关系。

那么这些信息来源于哪里呢？像 Linux 系统中的许多东西一样，来自一系列的文本文件。用户帐户 定义在 /etc/passwd 文件里面，用户组定义在 /etc/group 文件里面。当用户帐户和用户组创建以后， 这些文件随着文件 /etc/shadow 的变动而修改，文件 /etc/shadow 包含了关于用户密码的信息。 对于每个用户帐号，文件 /etc/passwd 定义了用户（登录）名，uid，gid，帐号的真实姓名，主目录， 和登录 shell。如果你查看一下文件 /etc/passwd 和文件 /etc/group 的内容，你会注意到除了普通 用户帐号之外，还有超级用户（uid 0）帐号，和各种各样的系统用户。

在下一章中，当我们讨论进程时，你会知道这些其他的“用户”是谁，实际上，他们相当忙碌。

然而许多像 Unix 的系统会把普通用户分配到一个公共的用户组中，例如 “users”，现在的 Linux 会创建一个独一无二的，只有一个成员的用户组，这个用户组与用户同名。这样使某种类型的 权限分配更容易些。

读取，写入，和执行

对于文件和目录的访问权力是根据读访问，写访问，和执行访问来定义的。如果我们看一下 ls 命令的输出结果，我们能得到一些线索，这是怎样实现的：

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me   me   0 2008-03-06 14:52 foo.txt
```

列表的前十个字符是文件的属性。这十个字符的第一个字符表明文件类型。下表是你可能经常看到 的文件类型（还有其它的，不常见类型）：

表10-1: 文件类型

属性	文件类型
-	一个普通文件
d	一个目录
l	一个符号链接。注意对于符号链接文件，剩余的文件属性总是 "rwxrwxrwx"，而且都是 虚拟值。真正的文件属性是指符号链接所指向的文件的属性。
c	一个字符设备文件。这种文件类型是指按照字节流，来处理数据的设备。 比如说终端机，或者调制解调器
b	一个块设备文件。这种文件类型是指按照数据块，来处理数据的设备，例如一个硬盘，或者 CD-ROM 盘。

剩下的九个字符，叫做文件模式，代表着文件所有者，文件组所有者，和其他人的读，写，执行权限。

当设置文件模式后，r，w，x 模式属性对文件和目录会产生以下影响：

chmod － 更改文件模式

更改文件或目录的模式（权限），可以利用 chmod 命令。注意只有文件的所有者或者超级用户才 能更改文件或目录的模式。chmod 命令支持两种不同的方法来改变文件模式：八进制数字表示法，或 符号表示法。首先我们讨论一下八进制数字表示法。

> 究竟什么是八进制？

> 八进制（以8为基数），和她的亲戚，十六进制（以16为基数）都是数字系统，通常 被用来表示计算机中的数字。我们人类，因为这个事实（或者至少大多数人）天生具有 十个手指，利用以10为基数的数字系统来计数。计算机，从另一方面讲，生来只有一个 手指，因此它以二进制（以2为基数）来计数。它们的数字系统只有两个数值，0和1。 因此在二进制中，计数看起来像这样：
>
> 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...
>
> 在八进制中，逢八进一，用数字0到7来计数，像这样：
>
> 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...
>
> 十六进制中，使用数字0到9，加上大写字母"A"到"F"来计数，逢16进一：
>
> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...
>
> 虽然我们能知道二进制的意义（因为计算机只有一个手指），但是八进制和十六进制有什么 用处呢？ 答案是为了人类的便利。许多时候，在计算机中，一小部分数据以二进制的形式表示。 以 RGB 颜色为例来说明。大多数的计算机显示器，每个像素由三种颜色组成：8位红色，8位绿色， 8位蓝色。这样，一种可爱的中蓝色就由24位数字来表示：
>
> 010000110110111111001101
>
> 我不认为你每天都喜欢读写这类数字。另一种数字系统对我们更有帮助。每个十六进制 数字代表四个二进制。在八进制中，每个数字代表三个二进制数字。那么代表中蓝色的24位 二进制能够压缩成6位十六进制数：
>
> 436FCD
>
> 因为十六进制中的两个数字对应二进制的8位数字，我们可以看到”43“代表红色，“6F” 代表绿色，“CD”代表蓝色。
>
> 现在，十六进制表示法（经常叫做“hex”）比八进制更普遍，但是我们很快会看到，用八进制 来表示3个二进制数非常有用处...

通过八进制表示法，我们使用八进制数字来设置所期望的权限模式。因为每个八进制数字代表了 3个二进制数字，这种对应关系，正好映射到用来存储文件模式所使用的方案上。下表展示了 我们所要表达的意思：

Octal	Binary	File Mode
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

通过使用3个八进制数字，我们能够设置文件所有者，用户组，和其他人的权限：

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0  2008-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me    0  2008-03-06 14:52 foo.txt
```

通过传递参数 “600”，我们能够设置文件所有者的权限为读写权限，而删除用户组和其他人的所有 权限。虽然八进制到二进制的映射看起来不方便，但通常只会用到一些常见的映射关系： 7 (rwx)， 6 (rw-)，5 (r-x)， 4 (r--)， and 0 (---)。

chmod 命令支持一种符号表示法，来指定文件模式。符号表示法分为三部分：更改会影响谁， 要执行哪个操作，要设置哪种权限。通过字符“u”，“g”，“o”，and “a”的组合来指定 要影响的对象，如下所示：

表10-4: chmod 命令符号表示法

u	“user”的简写，意思是文件或目录的所有者。
g	用户组。
o	“others” 的简写，意思是其他所有的人。
a	“all” 的简写，是“u”，“g”，和 “o” 三者的联合。

如果没有指定字符，则假定使用“all”。执行的操作可能是一个“＋”字符，表示加上一个权限， 一个“－”，表示删掉一个权限，或者是一个“＝”，表示只有指定的权限可用，其它所有的权限被删除。

权限由“r”, “w”, and “x”来指定。这里是一些符号表示法的实例：

表10-5: chmod 符号表示法实例

u+x	为文件所有者添加可执行权限。
u-x	删除文件所有者的可执行权限。
+x	为文件所有者，用户组，和其他所有人添加可执行权限。等价于 a+x。
o-rw	除了文件所有者和用户组，删除其他人的读权限和写权限。
go=rw	给群组的主人和任意文件拥有者的人读写权限。如果群组的主人或全局之前已经有了执行的权限，他们将被移除。
u+x,go=rw	给文件拥有者执行权限并给组和其他人读和执行的权限。多种设定可以用逗号分开。

一些人喜欢使用八进制表示法，而另些人真正地喜欢符号表示法。符号表示法的优点是， 允许你设置文件模式的单个组成部分的属性，而没有影响其他的部分。

看一下 chmod 命令的手册页，可以得到更详尽的信息和 chmod 命令的各个选项。要注意 “--recursive” 选项： 它可以同时作用于文件和目录，所以它并不是如我们期望的那么有用处，因为我们很少希望文件和 目录拥有同样的权限。

借助 GUI 来设置文件模式

现在我们已经知道了怎样设置文件和目录的权限，这样我们就可以更好的理解 GUI 中的设置 权限对话框。在 Nautilus (GNOME)和 Konqueror (KDE)中，右击一个文件或目录图标将会弹出一个属性对话框。下面这个例子来自 KDE 3.5：

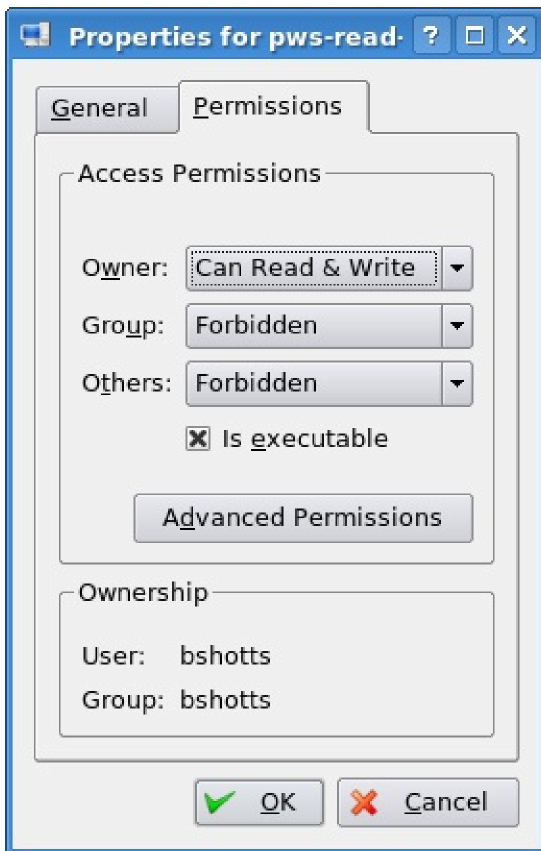


图2: KDE 3.5 文件属性对话框

从这个对话框中，我们看到可以设置文件所有者，用户组，和其他人的访问权限。 在 KDE 中，右击 "Advanced Permissions"按钮，会打开另一个对话框，这个对话框允许 你单独设置各个模式属性。这也可以通过命令行来理解！

umask － 设置默认权限

当创建一个文件时，umask 命令控制着文件的默认权限。umask 命令使用八进制表示法来表达 从文件模式属性中删除一个位掩码。大家看下面的例子：

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me   me   0 2008-03-06 14:53 foo.txt
```

首先，删除文件 foo.txt，以此确定我们从新开始。下一步，运行不带参数的 umask 命令， 看一下当前的掩码值。响应的数值是0002（0022是另一个常用值），这个数值是掩码的八进制 表示形式。下一步，我们创建文件 foo.txt，并且保留它的权限。

我们可以看到文件所有者和用户组都得到读权限和写权限，而其他人只是得到读权限。 其他人没有得到写权限的原因是由掩码值决定的。重复我们的实验，这次自己设置掩码值：

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me   me    0 2008-03-06 14:58 foo.txt
```

当掩码设置为0000（实质上是关掉它）之后，我们看到其他人能够读写文件。为了弄明白这是 怎么回事，我们需要看一下掩码的八进制形式。把掩码展开成二进制形式，然后与文件属性 相比较，看看有什么区别：

Original file mode	--- rw- rw- rw-
Mask	000 000 000 010
Result	--- rw- rw- r--

此刻先忽略掉开头的三个零（我们一会儿再讨论），注意掩码中若出现一个数字1，则 删除文件模式中和这个1在相同位置的属性，在这是指其他人的写权限。这就是掩码要完成的 任务。掩码的二进制形式中，出现数字1的位置，相应地关掉一个文件模式属性。看一下 掩码0022的作用：

Original file mode	--- rw- rw- rw-
Mask	000 000 010 010
Result	--- rw- r-- r--

又一次，二进制中数字1出现的位置，相对应的属性被删除。再试一下其它的掩码值（一些带数字7的），习惯于掩码的工作原理。当你实验完成之后，要记得清理现场：

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

大多数情况下，你不必修改掩码值，系统提供的默认掩码值就很好了。然而，在一些高 安全级别下，你要能控制掩码值。

> 一些特殊权限
>
> 虽然我们通常看到一个八进制的权限掩码用三位数字来表示，但是从技术层面上来讲， 用四位数字来表示它更确切些。为什么呢？因为，除了读取，写入，和执行权限之外，还有 其它的，较少用到的权限设置。
>
> 其中之一是 setuid 位（八进制4000)。当应用到一个可执行文件时，它把有效用户 ID 从真正的用户（实际运行程序的用户）设置成程序所有者的 ID。这种操作通常会应用到 一些由超级用户所拥有的程序。当一个普通用户运行一个程序，这个程序由根用户(root) 所有，并且设置了 setuid 位，这个程序运行时具有超级用户的特权，这样程序就可以 访问普通用户禁止访问的文件和目录。很明显，因为这会引起安全方面的问题，所有可以 设置 setuid 位的程序个数，必须控制在绝对小的范围内。
>
> 第二个是 setgid 位（八进制2000），这个相似于 setuid 位，把有效用户组 ID 从真正的 用户组 ID 更改为文件所有者的组 ID。如果设置了一个目录的 setgid 位，则目录中新创建的文件 具有这个目录用户组的所有权，而不是文件创建者所属用户组的所有权。对于共享目录来说， 当一个普通用户组中的成员，需要访问共享目录中的所有文件，而不管文件所有者的主用户组时， 那么设置 setgid 位很有用处。
>
> 第三个是 sticky 位（八进制1000）。这个继承于 Unix，在 Unix 中，它可能把一个可执行文件 标志为“不可交换的”。在 Linux 中，会忽略文件的 sticky 位，但是如果一个目录设置了 sticky 位， 那么它能阻止用户删除或重命名文件，除非用户是这个目录的所有者，或者是文件所有者，或是 超级用户。这个经常用来控制访问共享目录，比方说/tmp。
>
> 这里有一些例子，使用 chmod 命令和符号表示法，来设置这些特殊的权限。首先， 授予一个程序 setuid 权限。
>
> chmod u+s program
>
> 下一步，授予一个目录 setgid 权限：
>
> chmod g+s dir
>
> 最后，授予一个目录 sticky 权限：
>
> chmod +t dir
>
> 当浏览 ls 命令的输出结果时，你可以确认这些特殊权限。这里有一些例子。首先，一个程序被设置为setuid属性：
>
> -rwsr-xr-x

> 具有 setgid 属性的目录：
>
> drwxrwsr-x
>
> 设置了 sticky 位的目录：
>
> drwxrwxrwt

更改身份

在不同的时候，我们会发现很有必要具有另一个用户的身份。经常地，我们想要得到超级 用户特权，来执行一些管理任务，但是也有可能"变为"另一个普通用户，比如说测试一个帐号。 有三种方式，可以拥有多重身份：

1. 注销系统并以其他用户身份重新登录系统。

2. 使用 su 命令。

3. 使用 sudo 命令。

我们将跳过第一种方法，因为我们知道怎样使用它，并且它缺乏其它两种方法的方便性。 在我们自己的 shell 会话中，su 命令允许你，假定为另一个用户的身份，以这个用户的 ID 启动一个新的 shell 会话，或者是以这个用户的身份来发布一个命令。sudo 命令允许一个管理员 设置一个叫做 /etc/sudoers 的配置文件，并且定义了一些具体命令，在假定的身份下，特殊用户 可以执行这些命令。选择使用哪个命令，很大程度上是由你使用的 Linux 发行版来决定的。 你的发行版可能这两个命令都包含，但系统配置可能会偏袒其中之一。我们先介绍 su 命令。

su－以其他用户身份和组 ID 运行一个 shell

su 命令用来以另一个用户的身份来启动 shell。这个命令语法看起来像这样：

```
su [-[l]] [user]
```

如果包含"-l"选项，那么会为指定用户启动一个需要登录的 shell。这意味着会加载此用户的 shell 环境， 并且工作目录会更改到这个用户的主目录。这通常是我们所需要的。如果不指定用户，那么就假定是 超级用户。注意（不可思议地），选项"-l"可以缩写为"-"，这是经常用到的形式。启动超级用户的 shell， 我们可以这样做：

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

按下回车符之后，shell 提示我们输入超级用户的密码。如果密码输入正确，出现一个新的 shell 提示符， 这表明这个 shell 具有超级用户特权（提示符的末尾字符是"#"而不是"$"），并且当前工作目录是超级用户的主目录 （通常是 /root）。一旦进入一个新的 shell，我们能执行超级用户所使用的命令。当工作完成后， 输入"exit"，则返回到原来的 shell:

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

以这样的方式使用 su 命令，也可以只执行单个命令，而不是启动一个新的可交互的 shell：

```
su -c 'command'
```

使用这种模式，命令传递到一个新 shell 中执行。把命令用单引号引起来很重要，因为我们不想 命令在我们的 shell 中展开，但需要在新 shell 中展开。

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root    754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

sudo－以另一个用户身份执行命令

sudo 命令在很多方面都相似于 su 命令，但是 sudo 还有一些非常重要的功能。管理员能够配置 sudo 命令，从而允许一个普通用户以不同的身份（通常是超级用户），通过一种非常可控的方式 来执行命令。尤其是，只有一个用户可以执行一个或多个特殊命令时，（更体现了 sudo 命令的方便性）。 另一个重要差异是 sudo 命令不要求超级用户的密码。使用 sudo 命令时，用户使用他/她自己的密码 来认证。比如说，例如，sudo 命令经过配置，允许我们运行一个虚构的备份程序，叫做 "backup_script"， 这个程序要求超级用户权限。通过 sudo 命令，这个程序会像这样运行：

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

按下回车键之后，shell 提示我们输入我们的密码（不是超级用户的）。一旦认证完成，则执行 具体的命令。su 和 sudo 之间的一个重要区别是 sudo 不会重新启动一个 shell，也不会加载另一个 用户的 shell 运行环境。这意味者命令不必用单引号引起来。注意通过指定各种各样的选项，这 种行为可以被推翻。详细信息，阅读 sudo 手册页。

想知道 sudo 命令可以授予哪些权限，使用"-l"选项，列出所有权限：

```
[me@linuxbox ~]$ sudo -l
User me may run the following commands on this host:
(ALL) ALL
```

> *Ubuntu 与 sudo*
>
> 普通用户经常会遇到这样的问题，怎样完成某些需要超级用户权限的任务。这些任务 包括安装和更新软件，编辑系统配置文件，和访问设备。在 Windows 世界里，这些任务是 通过授予用户管理员权限来完成的。这允许用户执行这些任务。然而，这也会导致用户所 执行的程序拥有同样的能力。在大多数情况下，这是我们所期望的，但是它也允许 malware （恶意软件），比方说电脑病毒，自由地支配计算机。
>
> 在 Unix 世界中，由于 Unix 是多用户系统，所以在普通用户和管理员之间总是存在很大的 差别。Unix 采取的方法是只有在需要的时候，才授予普通用户超级用户权限。这样，普遍会 用到 su 和 sudo 命令。
>
> 几年前，大多数的 Linux 发行版都依赖于 su 命令，来达到目的。su 命令不需要 sudo 命令 所要求的配置，su 命令拥有一个 root 帐号，是 Unix 中的传统。但这会引起问题。所有用户 会企图以 root 用户帐号来操纵系统。事实上，一些用户专门以 root 用户帐号来操作系统， 因为这样做，的确消除了所有那些讨厌的“权限被拒绝”的消息。相比于 Windows 系统安全性而言， 这样做，你就削弱了 Linux 系统安全性能。不是一个好主意。
>
> 当引进 Ubuntu 的时候，它的创作者们采取了不同的策略。默认情况下，Ubuntu 不允许用户登录 到 root 帐号（因为不能为 root 帐号设置密码），而是使用 sudo 命令授予普通用户超级用户权限。 通过 sudo 命令，最初的用户可以拥有超级用户权限，也可以授予随后的用户帐号相似的权力。

chown－更改文件所有者和用户组

chown 命令被用来更改文件或目录的所有者和用户组。使用这个命令需要超级用户权限。chown 命令 的语法看起来像这样：

```
chown [owner][:[group]] file...
```

chown 命令可以更改文件所有者和/或文件用户组，依据于这个命令的第一个参数。这里有 一些例子：

表10-6: chown 参数实例

参数	结果
bob	把文件所有者从当前属主更改为用户 bob。
bob:users	把文件所有者改为用户 bob，文件用户组改为用户组 users。
:admins	把文件用户组改为组 admins，文件所有者不变。
bob:	文件所有者改为用户 bob，文件用户组改为，用户 bob 登录系统时，所属的用户组。

比方说，我们有两个用户，janet，拥有超级用户访问权限，而 tony 没有。用户 jant 想要从 她的主目录复制一个文件到用户 tony 的主目录。因为用户 jant 想要 tony 能够编辑这个文件， janet 把这个文件的所有者更改为 tony:

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1 root  root 8031 2008-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1 tony  tony 8031 2008-03-20 14:30 /home/tony/myfile.txt
```

这里，我们看到用户 janet 把文件从她的目录复制到 tony 的主目录。下一步，janet 把文件所有者 从 root（使用 sudo 命令的原因）改到 tony。通过在第一个参数中使用末尾的":"字符，janet 同时把 文件用户组改为 tony 登录系统时，所属的用户组，碰巧是用户组 tony。

注意，第一次使用 sudo 命令之后，为什么（shell）没有提示 janet 输入她的密码？这是因为，在 大多数的配置中，sudo 命令会相信你几分钟，直到计时结束。

chgrp－更改用户组所有权

在旧版 Unix 系统中，chown 命令只能更改文件所有权，而不是用户组所有权。为了达到目的， 使用一个独立的命令，chgrp 来完成。除了限制多一点之外，chgrp 命令与 chown 命令使用起来很相似。

练习使用权限

到目前为止，我们已经知道了，权限这类东西是怎样工作的，现在是时候炫耀一下了。我们 将展示一个常见问题的解决方案，这个问题是如何设置一个共享目录。假想我们有两个用户， 他们分别是 "bill" 和 "karen"。他们都有音乐 CD 收藏品，也愿意设置一个共享目录，在这个 共享目录中，他们分别以 Ogg Vorbis 或 MP3 的格式来存储他们的音乐文件。通过 sudo 命令， 用户 bill 具有超级用户访问权限。

我们需要做的第一件事，是创建一个以 bill 和 karen 为成员的用户组。使用图形化的用户管理工具， bill 创建了一个叫做 music 的用户组，并且把用户 bill 和 karen 添加到用户组 music 中：



图3: 用 GNOME 创建一个新的用户组

下一步，bill 创建了存储音乐文件的目录：

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
password:
```

因为 bill 正在他的主目录之外操作文件，所以需要超级用户权限。这个目录创建之后，它具有 以下所有权和权限：

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2008-03-21 18:05 /usr/local/share/Music
```

正如我们所见到的，这个目录由 root 用户拥有，并且具有权限755。为了使这个目录共享，允许（用户 karen）写入，bill 需要更改目录用户组所有权和权限：

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2008-03-21 18:05 /usr/local/share/Music
```

那么这是什么意思呢？ 它的意思是，现在我们拥有一个目录，/usr/local/share/Music，这个目录由 root 用户拥有，并且 允许用户组 music 读取和写入。用户组 music 有两个成员 bill 和 karen，这样 bill 和 karen 能够在目录 /usr/local/share/Music 中创建文件。其他用户能够列出目录中的内容，但是不能在其中创建文件。

但是我们仍然会遇到问题。通过我们目前所拥有的权限，在 Music 目录中创建的文件，只具有用户 bill 和 karen 的普通权限：

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill    bill    0 2008-03-24 20:03 test_file
```

实际上，存在两个问题。第一个，系统中默认的掩码值是0022，这会禁止用户组成员编辑属于同 组成员的文件。如果共享目录中只包含文件，这就不是个问题，但是因为这个目录将会存储音乐， 通常音乐会按照艺术家和唱片的层次结构来组织分类。所以用户组成员需要在同组其他成员创建的 目录中创建文件和目录。我们将把用户 bill 和 karen 使用的掩码值改为0002。

第二个问题是，用户组成员创建的文件和目录的用户组，将会设置为用户的主要组，而不是用户组 music。通过设置此目录的 setgid 位来解决这个问题：

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2008-03-24 20:03 /usr/local/share/Music
```

现在测试一下，看看是否新的权限解决了这个问题。bill 把他的掩码值设为0002，删除 先前的测试文件，并创建了一个新的测试文件和目录：

```
[bill@linuxbox ~]$ umask 0002

[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file

[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2008-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music 0 2008-03-24 20:22 test_file
[bill@linuxbox ~]$
```

现在，创建的文件和目录都具有正确的权限，允许用户组 music 的所有成员在目录 Music 中创建 文件和目录。

剩下一个问题是关于 umask 命令的。umask 命令设置的掩码值只能在当前 shell 会话中生效，若当前 shell 会话结束后，则必须重新设置。在这本书的第三部分，我们将看一下，怎样使掩码值永久生效。

更改用户密码

这一章最后一个话题，我们将讨论自己帐号的密码（和其他人的密码，如果你具有超级用户权限）。使用 passwd 命令，来设置或更改用户密码。命令语法如下所示：

```
passwd [user]
```

只要输入 passwd 命令，就能更改你的密码。shell 会提示你输入你的旧密码和你的新密码：

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
```

passwd 命令将会试着强迫你使用“强”密码。这意味着，它会拒绝接受太短的密码，与先前相似的密码， 字典中的单词作为密码，或者是太容易猜到的密码：

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

如果你具有超级用户权限，你可以指定一个用户名作为 passwd 命令的参数，这样可以设置另一个 用户的密码。还有其它的 passwd 命令选项对超级用户有效，允许帐号锁定，密码失效，等等。详细内容，参考 passwd 命令的手册页。

拓展阅读

- Wikipedia 上面有一篇关于 malware（恶意软件）好文章：

http://en.wikipedia.org/wiki/Malware

还有一系列的命令行程序，可以用来创建和维护用户和用户组。更多信息，查看以下命令的手册页：

- adduser
- useradd
- groupadd

进程

通常，现在的操作系统都支持多任务，意味着操作系统（给用户）造成了一种假象，(让用户觉得) 它同时能够做多件事情，事实上，它是快速地轮换执行这些任务的。Linux 内核通过使用进程，来 管理多任务。通过进程，Linux 安排不同的程序等待使用 CPU。

有时候，计算机变得呆滞，运行缓慢，或者一个应用程序停止响应。在这一章中，我们将看一些 可用的命令行工具，这些工具帮助我们查看程序的执行状态，以及怎样终止行为不当的进程。

这一章将介绍以下命令：

- ps – 报告当前进程快照
- top – 显示任务
- jobs – 列出活跃的任务
- bg – 把一个任务放到后台执行
- fg – 把一个任务放到前台执行
- kill – 给一个进程发送信号
- killall – 杀死指定名字的进程
- shutdown – 关机或重启系统

进程是怎样工作的

当系统启动的时候，内核先把一些它自己的程序初始化为进程，然后运行一个叫做 init 的程序。init， 依次地，再运行一系列的称为 init 脚本的 shell 脚本（位于/etc），它们可以启动所有的系统服务。 其中许多系统服务以守护（daemon）程序的形式实现，守护程序仅在后台运行，没有任何用户接口。 这样，即使我们没有登录系统，至少系统也在忙于执行一些例行事务。

一个程序可以发动另一个程序，这个事实在进程方案中，表述为一个父进程创建了一个子进程。

内核维护每个进程的信息，以此来保持事情有序。例如，系统分配给每个进程一个数字，这个数字叫做 进程 ID 或 PID。PID 号按升序分配，init 进程的 PID 总是1。内核也对分配给每个进程的内存进行跟踪。 像文件一样，进程也有所有者和用户 ID，有效用户 ID，等等。

查看进程

查看进程，最常使用地命令（有几个命令）是 ps。ps 程序有许多选项，它最简单地使用形式是这样的：

```
[me@linuxbox ~]$ ps
PID TTY           TIME CMD
5198 pts/1    00:00:00 bash
10129 pts/1   00:00:00 ps
```

上例中，列出了两个进程，进程 5198 和进程 10129，各自代表命令 bash 和 ps。正如我们所看到的， 默认情况下，ps 不会显示很多进程信息，只是列出与当前终端会话相关的进程。为了得到更多信息， 我们需要加上一些选项，但是在这样做之前，我们先看一下 ps 命令运行结果的其它字段。 TTY 是 "Teletype" 的简写，是指进程的控制终端。这里，Unix 展示它的年龄。TIME 字段表示 进程所消耗的 CPU 时间数量。正如我们所看到的，这两个进程使计算机工作起来很轻松。

如果给 ps 命令加上选项，我们可以得到更多关于系统运行状态的信息：

```
[me@linuxbox ~]$ ps x
PID TTY   STAT   TIME COMMAND
2799 ?    Ssl    0:00 /usr/libexec/bonobo-activation-server –ac
2820 ?    Sl     0:01 /usr/libexec/evolution-data-server-1.10 --

and many more...
```

加上 "x" 选项（注意没有开头的 "-" 字符），告诉 ps 命令，展示所有进程，不管它们由什么 终端（如果有的话）控制。在 TTY 一栏中出现的 "?"，表示没有控制终端。使用这个 "x" 选项，可以 看到我们所拥有的每个进程的信息。

因为系统中正运行着许多进程，所以 ps 命令的输出结果很长。这经常很有帮助，要是把 ps 的输出结果 管道到 less 命令，借助 less 工具，更容易浏览。一些选项组合也会产生很长的输出结果，所以最大化 终端仿真器窗口，也是一个好主意。

输出结果中，新添加了一栏，标题为 STAT 。STAT 是 "state" 的简写，它揭示了进程当前状态：

表11-1: 进程状态

状态	意义
R	运行。这意味着，进程正在运行或准备运行。
S	正在睡眠。 进程没有运行，而是，正在等待一个事件， 比如说，一个按键或者网络数据包。
D	不可中断睡眠。进程正在等待 I/O，比方说，一个磁盘驱动器的 I/O。
T	已停止. 已经指示进程停止运行。稍后介绍更多。
Z	一个死进程或“僵尸”进程。这是一个已经终止的子进程，但是它的父进程还没有清空它。 （父进程没有把子进程从进程表中删除）
<	一个高优先级进程。这可能会授予一个进程更多重要的资源，给它更多的 CPU 时间。 进程的这种属性叫做 niceness。具有高优先级的进程据说是不好的（less nice）， 因为它占用了比较多的 CPU 时间，这样就给其它进程留下很少时间。
N	低优先级进程。 一个低优先级进程（一个“好”进程）只有当其它高优先级进程执行之后，才会得到处理器时间。

进程状态信息之后，可能还跟随其他的字符。这表示各种外来进程的特性。详细信息请看 ps 手册页。

另一个流行的选项组合是 "aux"（不带开头的"-"字符）。这会给我们更多信息：

```
[me@linuxbox ~]$ ps aux
USER   PID  %CPU  %MEM    VSZ    RSS  TTY   STAT   START   TIME  COMMAND
root     1   0.0   0.0    2136    644  ?     Ss     Mar05   0:31  init
root     2   0.0   0.0       0      0  ?     S&lt;     Mar05   0:00  [kt]

and many more...
```

这个选项组合，能够显示属于每个用户的进程信息。使用这个选项，可以唤醒 “BSD 风格” 的输出结果。 Linux 版本的 ps 命令，可以模拟几个不同 Unix 版本中的 ps 程序的行为。通过这些选项，我们得到 这些额外的列。

表11-2: BSD 风格的 ps 命令列标题

标题	意思
USER	用户 ID. 进程的所有者。
%CPU	以百分比表示的 CPU 使用率
%MEM	以百分比表示的内存使用率
VSZ	虚拟内存大小
RSS	进程占用的物理内存的大小，以千字节为单位。
START	进程运行的起始时间。若超过24小时，则用天表示。

用 top 命令动态查看进程

虽然 ps 命令能够展示许多计算机运行状态的信息，但是它只是提供，ps 命令执行时刻的机器状态快照。 为了看到更多动态的信息，我们使用 top 命令：

```
[me@linuxbox ~]$ top
```

top 程序连续显示系统进程更新的信息（默认情况下，每三分钟更新一次），"top"这个名字 来源于这个事实，top 程序是用来查看系统中“顶端”进程的。top 显示结果由两部分组成： 最上面是系统概要，下面是进程列表，以 CPU 的使用率排序。

```
top - 14:59:20 up 6:30, 2 users, load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running,  106 sleeping,    0 stopped,    2 zombie
Cpu(s): 0.7%us, 1.0%sy, 0.0%ni, 98.3%id, 0.0%wa, 0.0%hi, 0.0%si
Mem:   319496k total,   314860k used,   4636k free,   19392k buff
Swap:  875500k total,   149128k used,   726372k free,  114676k cach

 PID  USER       PR   NI   VIRT   RES   SHR  S %CPU  %MEM   TIME+    COMMAND
6244  me         39   19  31752  3124  2188  S  6.3   1.0   16:24.42 trackerd
....
```

其中系统概要包含许多有用信息。下表是对系统概要的说明：

表11-3: top 命令信息字段

行号	字段	意义
1	top	程序名。
	14:59:20	当前时间。
	up 6:30	这是正常运行时间。它是计算机从上次启动到现在所运行的时间。 在这个例子里，系统已经运行了六个半小时。
	2 users	有两个用户登录系统。
	load average:	加载平均值是指，等待运行的进程数目，也就是说，处于运行状态的进程个数， 这些进程共享 CPU。展示了三个数值，每个数值对应不同的时间周期。第一个是最后60秒的平均值， 下一个是前5分钟的平均值，最后一个是前15分钟的平均值。若平均值低于1.0，则指示计算机 工作不忙碌。
2	Tasks:	总结了进程数目和各种进程状态。
3	Cpu(s):	这一行描述了 CPU 正在执行的进程的特性。
	0.7%us	0.7% of the CPU is being used for user processes. 这意味着进程在内核之外。
	1.0%sy	1.0%的 CPU 时间被用于系统（内核）进程。
	0.0%ni	0.0%的 CPU 时间被用于"nice"（低优先级）进程。
	98.3%id	98.3%的 CPU 时间是空闲的。
	0.0%wa	0.0%的 CPU 时间来等待 I/O。
4	Mem:	展示物理内存的使用情况。
5	Swap:	展示交换分区（虚拟内存）的使用情况。

top 程序接受一系列从键盘输入的命令。两个最有趣的命令是 h 和 q。h，显示程序的帮助屏幕，q， 退出 top 程序。

两个主要的桌面环境都提供了图形化应用程序，来显示与 top 程序相似的信息 （和 Windows 中的任务管理器差别不多），但是我觉得 top 程序要好于图形化的版本， 因为它运行速度快，并且消费很少的系统资源。毕竟，我们的系统监测程序不能成为 系统怠工的源泉，而这是我们试图追踪的信息。

控制进程

现在我们可以看到和监测进程，然后得到一些对它们的控制权。为了我们的实验，我们将使用 一个叫做 xlogo 的小程序，作为我们的实验品。这个 xlogo 程序是 X 窗口系统 （底层引擎使图形界面显示在屏幕上）提供的实例程序，这个实例简单地显示一个大小可调的 包含 X 标志的窗口。首先，我们需要知道测试的主题：

```
[me@linuxbox ~]$ xlogo
```

命令执行之后，一个包含 X 标志的小窗口应该出现在屏幕的某个位置上。在一些系统中，xlogo 命令 会打印一条警告信息，但是不用理会它。

小贴士：如果你的系统不包含 xlogo 程序，试着用 gedit 或者 kwrite 来代替。

通过调整它的窗口大小，我们能够证明 xlogo 程序正在运行。如果这个标志以新的尺寸被重画， 则这个程序正在运行。

注意，为什么我们的 shell 提示符还没有返回？这是因为 shell 正在等待这个程序结束， 就像到目前为止我们用过的其它所有程序一样。如果我们关闭 xlogo 窗口，shell 提示符就返回了。

中断一个进程

我们再运行 xlogo 程序一次，观察一下发生了什么事。首先，执行 xlogo 命令，并且 证实这个程序正在运行。下一步，回到终端窗口，按下 Ctrl-c。

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

在一个终端中，输入 Ctrl-c，中断一个程序。这意味着，我们礼貌地要求终止这个程序。 输入 Ctrl-c 之后，xlogo 窗口关闭，shell 提示符返回。

通过这个技巧，许多（但不是全部）命令行程序可以被中断。

把一个进程放置到后台(执行)

比方说，我们想让 shell 提示符返回，却没有终止 xlogo 程序。为达到这个目的，我们把 这个程序放到后台执行。把终端看作是一个有前台（表层放置可见的事物，像 shell 提示符） 和后台（表层之下放置隐藏的事物）（的设备）。启动一个程序，让它立即在后台 运行，我们在程序命令之后，加上"&"字符：

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

执行命令之后，这个 xlogo 窗口出现，并且 shell 提示符返回，同时打印一些有趣的数字。 这条信息是 shell 特性的一部分，叫做工作控制。通过这条信息，shell 告诉我们，已经启动了 工作号为1（“［1］”），PID 为28236的程序。如果我们运行 ps 命令，可以看到我们的进程：

```
[me@linuxbox ~]$ ps
  PID TTY         TIME   CMD
10603 pts/1   00:00:00   bash
28236 pts/1   00:00:00   xlogo
28239 pts/1   00:00:00   ps
```

工作控制，这个 shell 功能可以列出从终端中启动的任务。执行 jobs 命令，我们可以看到这个输出列表：

```
[me@linuxbox ~]$ jobs
[1]+ Running            xlogo &
```

结果显示我们有一个任务，编号为“1”，它正在运行，并且这个任务的命令是 xlogo &。

进程返回到前台

一个在后台运行的进程对一切来自键盘的输入都免疫，也不能用 Ctrl-c 来中断它。使用 fg 命令，让一个进程返回前台执行：

```
[me@linuxbox ~]$ jobs
[1]+ Running        xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

fg 命令之后，跟随着一个百分号和工作序号（叫做 jobspec）。如果我们只有一个后台任务，那么 jobspec 是可有可无的。输入 Ctrl-c 来终止 xlogo 程序。

停止一个进程

有时候，我们想要停止一个进程，而没有终止它。这样会把一个前台进程移到后台等待。 输入 Ctrl-z，可以停止一个前台进程。让我们试一下。在命令提示符下，执行 xlogo 命令， 然后输入 Ctrl-z:

```
[me@linuxbox ~]$ xlogo
[1]+ Stopped                 xlogo
[me@linuxbox ~]$
```

停止 xlogo 程序之后，通过调整 xlogo 的窗口大小，我们可以证实这个程序已经停止了。 它看起来像死掉了一样。使用 fg 命令，可以恢复程序到前台运行，或者用 bg 命令把程序移到后台。

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

和 fg 命令一样，如果只有一个任务的话，jobspec 参数是可选的。

因为把一个进程从前台移到后台很方便，如果我们从命令行启动一个图形界面的程序，但是 忘记把它放到后台执行，即没有在命令后加上字符"&"，（也不用担心）。

为什么要从命令行启动一个图形界面程序呢？有两个原因。第一个，你想要启动的程序，可能 没有在窗口管理器的菜单中列出来（比方说 xlogo）。第二个，从命令行启动一个程序， 你能够看到一些错误信息，如果从窗口系统中运行程序的话，这些信息是不可见的。有时候， 一个程序不能从图形界面菜单中启动。这时候，应该从命令行中启动它。我们可能会看到 错误信息，这些信息揭示了问题所在。一些图形界面程序还有许多有意思并且有用的命令行选项。

kill 命令被用来“杀死”程序。这样我们就可以终止需要杀死的程序。这里有一个实例：

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+ Terminated               xlogo
```

首先，我们在后台启动 xlogo 程序。shell 打印出 jobspec 和这个后台进程的 PID。下一步，我们使用 kill 命令，并且指定我们想要终止的进程 PID。也可以用 jobspec（例如，“%1”）来代替 PID。

虽然这个命令很直接了当，但不仅仅这些。这个 kill 命令不是确切地“杀死”程序，而是给程序 发送信号。信号是操作系统与程序之间进行通信，所采用的几种方式中的一种。我们已经看到 信号，在使用 Ctrl-c 和 Ctrl-z 的过程中。当终端接受了其中一个按键组合后，它会给在前端运行 的程序发送一个信号。在使用 Ctrl-c 的情况下，会发送一个叫做 INT（中断）的信号；当使用 Ctrl-z 时，则发送一个叫做 TSTP（终端停止）的信号。程序，反过来，倾听信号的到来，当程序 接到信号之后，则做出响应。一个程序能够倾听和响应信号，这个事实允许一个程序做些事情， 比如，当程序接到一个终止信号时，它可以保存所做的工作。

通过 **kill** 命令给进程发送信号

kill 命令被用来给程序发送信号。它最常见的语法形式看起来像这样：

```
kill [-signal] PID...
```

如果在命令行中没有指定信号，那么默认情况下，发送 TERM（终止）信号。kill 命令被经常 用来发送以下命令：

表 11-4: 常用信号

编号	名字	含义
1	HUP	挂起。这是美好往昔的痕迹，那时候终端机通过电话线和调制解调器连接到 远端的计算机。这个信号被用来告诉程序，控制的终端机已经“挂起”。 通过关闭一个终端会话，可以说明这个信号的作用。发送这个信号到终端机上的前台程序，程序会终止。 许多守护进程也使用这个信号，来重新初始化。这意味着，当发送这个信号到一个守护进程后， 这个进程会重新启动，并且重新读取它的配置文件。Apache 网络服务器守护进程就是一个例子。
2	INT	中断。实现和 Ctrl-c 一样的功能，由终端发送。通常，它会终止一个程序。
9	KILL	杀死。这个信号很特别。鉴于进程可能会选择不同的方式，来处理发送给它的 信号，其中也包含忽略信号，这样呢，从不发送 Kill 信号到目标进程。而是内核立即终止 这个进程。当一个进程以这种方式终止的时候，它没有机会去做些“清理”工作，或者是保存劳动成果。 因为这个原因，把 KILL 信号看作杀手锏，当其它终止信号失败后，再使用它。
15	TERM	终止。这是 kill 命令发送的默认信号。如果程序仍然“活着”，可以接受信号，那么 这个信号终止。
18	CONT	继续。在停止一段时间后，进程恢复运行。
19	STOP	停止。这个信号导致进程停止运行，而没有终止。像 KILL 信号，它不被 发送到目标进程，因此它不能被忽略。

让我们实验一下 kill 命令：

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+ Hangup         xlogo
```

在这个例子里，我们在后台启动 xlogo 程序，然后通过 kill 命令，发送给它一个 HUP 信号。 这个 xlogo 程序终止运行，并且 shell 指示这个后台进程已经接受了一个挂起信号。在看到这条 信息之前，你可能需要多按几次 enter 键。注意，既可以用号码，也可以用名字，不过要在名字前面 加上字母“SIG”，来指定所要发送的信号。

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+ Hangup                  xlogo
```

重复上面的例子，试着使用其它的信号。记住，你也可以用 jobspecs 来代替 PID。

进程，和文件一样，拥有所有者，所以为了能够通过 kill 命令来给进程发送信号， 你必须是进程的所有者（或者是超级用户）。

除了上表列出的 kill 命令最常使用的信号之外，还有一些系统频繁使用的信号。以下是其它一些常用 信号列表：

表 11-5: 其它常用信号

编号	名字	含义
3	QUIT	退出
11	SEGV	段错误。如果一个程序非法使用内存，就会发送这个信号。也就是说， 程序试图写入内存，而这个内存空间是不允许此程序写入的。
20	TSTP	终端停止。当按下 Ctrl-z 组合键后，终端发送这个信号。不像 STOP 信号， TSTP 信号由目标进程接收，且可能被忽略。

28	WINCH	改变窗口大小。当改变窗口大小时，系统会发送这个信号。 一些程序，像 top 和 less 程序会响应这个信号，按照新窗口的尺寸，刷新显示的内容。

为了满足读者的好奇心，通过下面的命令可以得到一个完整的信号列表：

```
[me@linuxbox ~]$ kill -l
```

通过 **killall** 命令给多个进程发送信号

也有可能通过 killall 命令，给匹配特定程序或用户名的多个进程发送信号。下面是 killall 命令的语法形式：

```
killall [-u user] [-signal] name...
```

为了说明情况，我们将启动一对 xlogo 程序的实例，然后再终止它们：

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]- Terminated              xlogo
[2]+ Terminated              xlogo
```

记住，和 kill 命令一样，你必须拥有超级用户权限才能给不属于你的进程发送信号。

更多和进程相关的命令

因为监测进程是一个很重要的系统管理任务，所以有许多命令与它相关。玩玩下面几个命令：

表11-6: 其它与进程相关的命令

命令名	命令描述
pstree	输出一个树型结构的进程列表，这个列表展示了进程间父/子关系。
vmstat	输出一个系统资源使用快照，包括内存，交换分区和磁盘 I/O。 为了看到连续的显示结果，则在命令名后加上延时的时间（以秒为单位）。例如，“vmstat 5”。 终止输出，按下 Ctrl-c 组合键。
xload	一个图形界面程序，可以画出系统负载的图形。
tload	与 xload 程序相似，但是在终端中画出图形。使用 Ctrl-c，来终止输出。

shell 环境

正如我们之前所讨论到的，shell 在 shell 会话中维护着大量的信息，这些信息称为 (shell) 环境。 存储在 shell 环境中的数据被程序用来确定配置属性。然而大多数程序用配置文件来存储程序设置， 某些程序也会查找存储在 shell 环境中的数值来调整他们的行为。知道了这些，我们就可以用 shell 环境 来自定制 shell 经历。

在这一章，我们将用到以下命令：

- printenv – 打印部分或所有的环境变量
- set – 设置 shell 选项
- export — 导出环境变量，让随后执行的程序知道。
- alias – 创建命令别名

什么存储在环境变量中？

shell 在环境中存储了两种基本类型的数据，虽然对于 bash 来说，很大程度上这些类型是不可 辨别的。它们是环境变量和 shell 变量。Shell 变量是由 bash 存放的一很少数据，而环境变量基本上 就是其它的所有数据。除了变量，shell 也存储了一些可编程的数据，命名为别名和 shell 函数。我们 已经在第六章讨论了别名，而 shell 函数（涉及到 shell 脚本）将会在第五部分叙述。

检查环境变量

我们既可以用 bash 的内部命令 set，或者是 printenv 程序来查看什么存储在环境当中。set 命令可以 显示 shell 和环境变量两者，而 printenv 只是显示环境变量。因为环境变量内容列表相当长，所以最好 把每个命令的输出结果管道到 less 命令：

```
[me@linuxbox ~]$ printenv | less
```

执行以上命令之后，我们应该能得到类似以下内容：

我们所看到的是环境变量及其数值的列表。例如，我们看到一个叫做 USER 的变量，这个变量值是 “me”。printenv 命令也能够列出特定变量的数值：

```
[me@linuxbox ~]$ printenv USER
me
```

当使用没有带选项和参数的 set 命令时，shell 和环境变量二者都会显示，同时也会显示定义的 shell 函数。不同于 printenv 命令，set 命令的输出结果很礼貌地按照字母顺序排列：

```
[me@linuxbox ~]$ set | less
```

也可以通过 echo 命令来查看一个变量的内容，像这样：

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

如果 shell 环境中的一个成员既不可用 set 命令也不可用 printenv 命令显示，则这个变量是别名。 输入不带参数的 alias 命令来查看它们:

一些有趣的变量

shell 环境中包含相当多的变量，虽然你的 shell 环境可能不同于这里展示的，但是你可能会看到 以下变量在你的 shell 环境中：

```
KDE_MULTIHEAD=false
SSH_AGENT_PID=6666
HOSTNAME=linuxbox
GPG_AGENT_INFO=/tmp/gpg-PdOt7g/S.gpg-agent:6689:1
SHELL=/bin/bash
TERM=xterm
XDG_MENU_PREFIX=kde-
HISTSIZE=1000
XDG_SESSION_COOKIE=6d7b05c65846c3eaf3101b0046bd2b00-1208521990.996705
-1177056199
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/me/.gtkrc-2.0:/home/me/.kde/sh
are/config/gtkrc-2.0
GTK_RC_FILES=/etc/gtk/gtkrc:/home/me/.gtkrc:/home/me/.kde/share/confi
g/gtkrc
GS_LIB=/home/me/.fonts
WINDOWID=29360136
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
KDE_FULL_SESSION=true
USER=me
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01
:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:\*.cmd=00;32:\*.exe:
```

我们所看到的是环境变量及其数值的列表。例如，我们看到一个叫做 USER 的变量，这个变量值是 "me"。printenv 命令也能够列出特定变量的数值：

```
[me@linuxbox ~]$ printenv USER
me
```

当使用没有带选项和参数的 set 命令时，shell 和环境变量二者都会显示，同时也会显示定义的 shell 函数。不同于 printenv 命令，set 命令的输出结果很礼貌地按照字母顺序排列：

```
[me@linuxbox ~]$ set | less
```

也可以通过 echo 命令来查看一个变量的内容，像这样：

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

如果 shell 环境中的一个成员既不可用 set 命令也不可用 printenv 命令显示，则这个变量是别名。 输入不带参数的 alias 命令来查看它们:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

一些有趣的变量

shell 环境中包含相当多的变量，虽然你的 shell 环境可能不同于这里展示的，但是你可能会看到 以下变量在你的 shell 环境中：

表12-1: 环境变量

变量	内容

DISPLAY	如果你正在运行图形界面环境，那么这个变量就是你显示器的名字。通常，它是 ":0"，意思是由 X 产生的第一个显示器。
EDITOR	文本编辑器的名字。
SHELL	shell 程序的名字。
HOME	用户主目录。
LANG	定义了字符集以及语言编码方式。
OLD_PWD	先前的工作目录。
PAGER	页输出程序的名字。这经常设置为/usr/bin/less。
PATH	由冒号分开的目录列表，当你输入可执行程序名后，会搜索这个目录列表。
PS1	Prompt String 1. 这个定义了你的 shell 提示符的内容。随后我们可以看到，这个变量 内容可以全面地定制。
PWD	当前工作目录。
TERM	终端类型名。类似于 Unix 的系统支持许多终端协议；这个变量设置你的终端仿真器所用的协议。
TZ	指定你所在的时区。大多数类似于 Unix 的系统按照协调时间时 (UTC) 来维护计算机内部的时钟， 然后应用一个由这个变量指定的偏差来显示本地时间。
USER	你的用户名

如果缺失了一些变量，不要担心，这些变量会因发行版本的不同而不同。

如何建立 shell 环境？

当我们登录系统后， 启动 bash 程序，并且会读取一系列称为启动文件的配置脚本， 这些文件定义了默认的可供所有用户共享的 shell 环境。然后是读取更多位于我们自己主目录中 的启动文件，这些启动文件定义了用户个人的 shell 环境。精确的启动顺序依赖于要运行的 shell 会话 类型。有两种 shell 会话类型：一个是登录 shell 会话， 另一个是非登录 shell 会话。

登录 shell 会话会提示用户输入用户名和密码；例如，我们启动一个虚拟控制台会话。当我们在 GUI 模式下 运行终端会话时，非登录 shell 会话会出现。

登录 shell 会读取一个或多个启动文件，正如表12－2所示：

表12-2: 登录 shell 会话的启动文件

文件	内容
/etc/profile	应用于所有用户的全局配置脚本。
~/.bash_profile	用户私人的启动文件。可以用来扩展或重写全局配置脚本中的设置。
~/.bash_login	如果文件 ~/.bash_profile 没有找到，bash 会尝试读取这个脚本。
~/.profile	如果文件 ~/.bash_profile 或文件 ~/.bash_login 都没有找到，bash 会试图读取这个文件。 这是基于 Debian 发行版的默认设置，比方说 Ubuntu。

非登录 shell 会话会读取以下启动文件：

表12-3: 非登录 shell 会话的启动文件

文件	内容
/etc/bash.bashrc	应用于所有用户的全局配置文件。
~/.bashrc	用户私有的启动文件。可以用来扩展或重写全局配置脚本中的设置。

除了读取以上启动文件之外，非登录 shell 会话也会继承它们父进程的环境设置，通常是一个登录 shell。

浏览一下你的系统，看一看系统中有哪些启动文件。记住－因为上面列出的大多数文件名都以圆点开头 （意味着它们是隐藏文件），你需要使用带"-a"选项的 ls 命令。

在普通用户看来，文件 ~/.bashrc 可能是最重要的启动文件，因为它几乎总是被读取。非登录 shell 默认 会读取它，并且大多数登录 shell 的启动文件会以能读取 ~/.bashrc 文件的方式来书写。

一个启动文件的内容

如果我们看一下典型的 .bash_profile 文件（来自于 CentOS 4 系统），它看起来像这样：

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi
# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
```

以"#"开头的行是注释，shell 不会读取它们。它们在那里是为了方便人们阅读。第一件有趣的事情 发生在第四行，伴随着以下代码：

```
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi
```

这叫做一个 if 复合命令，我们将会在第五部分详细地介绍它，现在我们对它翻译一下：

```
If the file ~/.bashrc exists, then
read the ~/.bashrc file.
```

我们可以看到这一小段代码就是一个登录 shell 得到 .bashrc 文件内容的方式。在我们启动文件中， 下一件有趣的事与 PATH 变量有关系。

曾经是否感到迷惑 shell 是怎样知道到哪里找到我们在命令行中输入的命令的？例如，当我们输入 ls 后， shell 不会查找整个计算机系统，来找到 /bin/ls（ls 命令的绝对路径名），而是，它查找一个目录列表， 这些目录包含在 PATH 变量中。

PATH 变量经常（但不总是，依赖于发行版）在 /etc/profile 启动文件中设置，通过这些代码：

```
PATH=$PATH:$HOME/bin
```

修改 PATH 变量，添加目录 $HOME/bin 到目录列表的末尾。这是一个参数展开的实例， 参数展开我们在第八章中提到过。为了说明这是怎样工作的，试试下面的例子：

```
[me@linuxbox ~]$ foo="This is some"
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo="$foo text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

使用这种技巧，我们可以把文本附加到一个变量值的末尾。通过添加字符串 $HOME/bin 到 PATH 变量值 的末尾，则目录 $HOME/bin 就添加到了命令搜索目录列表中。这意味着当我们想要在自己的主目录下， 创建一个目录来存储我们自己的私人程序时，shell 已经给我们准备好了。我们所要做的事就是 把创建的目录叫做 bin，赶快行动吧。

注意：很多发行版默认地提供了这个 PATH 设置。一些基于 Debian 的发行版，例如 Ubuntu，在登录 的时候，会检测目录 ~/bin 是否存在，若找到目录则把它动态地加到 PATH 变量中。

最后，有下面一行代码：

```
export PATH
```

这个 export 命令告诉 shell 让这个 shell 的子进程可以使用 PATH 变量的内容。

修改 shell 环境

既然我们知道了启动文件所在的位置和它们所包含的内容，我们就可以修改它们来定制自己的 shell 环境。

我们应该修改哪个文件？

按照通常的规则，添加目录到你的 PATH 变量或者是定义额外的环境变量，要把这些更改放置到 .bash_profile 文件中（或者其替代文件中，根据不同的发行版。例如，Ubuntu 使用 .profile 文件）。 对于其它的更改，要放到 .bashrc 文件中。除非你是系统管理员，需要为系统中的所有用户修改 默认设置，那么则限定你只能对自己主目录下的文件进行修改。当然，有可能会更改 /etc 目录中的 文件，比如说 profile 文件，而且在许多情况下，修改这些文件也是明智的，但是现在，我们要 安全起见。

文本编辑器

为了编辑（例如，修改）shell 的启动文件，还有系统中大多数其它配置文件，我们使用一个叫做文本 编辑器的程序。文件编辑器是一个，在某些方面，类似于文字处理器的程序，比如说随着鼠标的移动， 它允许你在屏幕上编辑文字。只有一点，文本编辑器不同于文字处理器，就是它只能支持纯文本，并且 经常包含为便于写程序而设计的特性。文本编辑器是软件开发人员用来写代码，和系统管理原员用来管理 系统配置文件的重要工具。

Linux 系统有许多不同类型的文本编辑器可用；你的系统中可能已经安装了几个。为什么会有这么 多种呢？可能因为程序员喜欢编写它们，又因为程序员们会频繁地使用它们，所以程序员编写编辑器让 它们按照程序员自己的愿望工作。

文本编辑器分为两种基本类型：图形化的和基于文本的编辑器。GNOME 和 KDE 两者都包含一些流行的 图形编辑器。GNOME 自带了一个叫做 gedit 的编辑器，这个编辑器通常在 GNOME 菜单中称为"文本编辑器"。 KDE 通常自带了三种编辑器，分别是（按照复杂度递增的顺序排列）kedit，kwrite，kate。

有许多基于文本的编辑器。你将会遇到一些流行的编辑器，它们是 nano，vi，和 emacs。这个 nano 编辑器 是一个简单的，容易使用的编辑器，它是 pico 编辑器的替代物，pico 编辑器由 PINE 邮件套件提供。vi 编辑器 （在大多数 Linux 系统中被 vim 替代，vim 是 "Vi IMproved"的简写）是类似于 Unix 操作系统的传统编辑器。 vim 是我们下一章节的讨论对象。emacs 编辑器最初由 Richard Stallman 写成。emacs 是一个庞大的，多用途的， 可做任何事情的编程环境。虽然 emacs 很容易获取，但是大多数 Linux 系统很少默认安装它。

使用文本编辑器

所有的文本编辑器都可以通过在命令行中输入编辑器的名字，加上你所想要编辑的文件来唤醒。如果所 输入的文件名不存在，编辑器则会假定你想要创建一个新文件。下面是一个使用 gedit 的例子：

```
[me@linuxbox ~]$ gedit some_file
```

这条命令将会启动 gedit 文本编辑器，同时加载名为 "some_file" 的文件，如果这个文件存在的话。

所有的图形文本编辑器都相当不言自明的，所以我们在这里不会介绍它们。反之，我们将集中精力在 我们第一个基于文本的文本编辑器，nano。让我们启动 nano，并且编辑文件 .bashrc。但是在我们这样 做之前，先练习一些"安全计算"。当我们编辑一个重要的配置文件时，首先创建一个这个文件的备份 总是一个不错的主意。这样能避免我们在编辑文件时弄乱文件。创建文件 .bashrc 的备份文件，这样做：

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

备份文件的名字无关紧要，只要选择一个容易理解的文件名。扩展名 ".bak"，".sav"， ".old"，和 ".orig" 都是用来指示备份文

件的流行方法。哦，记住 cp 命令会默默地重写存在的文件。

现在我们有了一个备份文件，我们启动 nano 编辑器吧：

```
[me@linuxbox ~]$ nano .bashrc
```

一旦 nano 编辑器启动后，我们将会得到一个像下面一样的屏幕：

```
GNU nano 2.0.3
....
```

注意：如果你的系统中没有安装 nano 编辑器，你可以用一个图形化的编辑器代替。

这个屏幕由上面的标头，中间正在编辑的文件文本和下面的命令菜单组成。因为设计 nano 是为了 代替由电子邮件客户端提供的编辑器的，所以它相当缺乏编辑特性。在任一款编辑器中，你应该 学习的第一个命令是怎样退出程序。以 nano 为例，你输入 Ctrl-x 来退出 nano。在屏幕底层的菜单中 说明了这个命令。"^X" 表示法意思是 Ctrl-x。这是控制字符的常见表示法，许多程序都使用它。

第二个我们需要知道的命令是怎样保存我们的劳动成果。对于 nano 来说是 Ctrl-o。尽然我们 已经获得了这些知识，接下来我们准备做些编辑工作。使用下箭头按键和 / 或下翻页按键，移动 鼠标到文件的最后一行，然后添加以下几行到文件 .bashrc 中：

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

注意：你的发行版可能已经包含其中的一些行，但是复制没有任何伤害。

下表是所添加行的意义：

文本行	含义
umask 0002	设置掩码来解决共享目录的问题。
export HISTCONTROL=ignoredups	使得 shell 的历史记录功能忽略一个命令，如果相同的命令已被记录。
export HISTSIZE=1000	增加命令历史的大小，从默认的 500 行扩大到 1000 行。
alias l.='ls -d .* --color=auto'	创建一个新命令，叫做'l.'，这个命令会显示所有以点开头的目录名。
alias ll='ls -l --color=auto'	创建一个叫做'll'的命令，这个命令会显示长格式目录列表。

正如我们所看到的，我们的许多附加物意思直觉上并不是明显的，所以添加注释到我们的文件 .bashrc 中是 一个好主意，可以帮助人们理解。使用编辑器，更改我们的附加物，让它们看起来像这样：

```
# Change umask to make directory sharing easier
umask 0002
 # Ignore duplicates in command history and increase
 # history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000
 # Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

啊，看起来好多了! 当我们完成修改后，输入 Ctrl-o 来保存我们修改的 .bashrc 文件，输入 Ctrl-x 退出 nano。

> 为什么注释很重要？
>
> 不管什么时候你修改配置文件时，给你所做的更改加上注释都是一个好主意。的确，明天你会 记得你修改了的内容，但是六个月之后会怎样呢？帮自己一个忙，加上一些注释吧。当你意识 到这一点后，对你所做的修改做个日志是个不错的主意。
>
> Shell 脚本和 bash 启动文件都使用 "#" 符号来开始注释。其它配置文件可能使用其它的符号。 大多数配置文件都有注释。把它们作为指南。
>
> 你会经常看到配置文件中的一些行被注释掉，以此防止它们被受影响的程序使用。这样做 是为了给读者在可能的配置选项方面一些建议，或者给出正确的配置语法实例。例如，Ubuntu 8.04 中的 .bashrc 文件包含这些行：
>
> ```
> # some more ls aliases
> #alias ll='ls -l'
> #alias la='ls -A'
> #alias l='ls -CF'
> ```
>
> 最后三行是有效的被注释掉的别名定义。如果你删除这三行开头的 "#" 符号，此技术程称为 uncommenting (不注释)，这样你就会激活这些别名。相反地，如果你在一行的开头加上 "#" 符号， 你可以注销掉这一行，但会保留它所包含的信息。

激活我们的修改

我们对于文件 .bashrc 的修改不会生效，直到我们关闭终端会话，再重新启动一个新的会话， 因为 .bashrc 文件只是在刚开始启动终端会话时读取。然而，我们可以强迫 bash 重新读取修改过的 .bashrc 文件，使用下面的命令：

```
[me@linuxbox ~]$ source .bashrc
```

运行上面命令之后，我们就应该能够看到所做修改的效果了。试试其中一个新的别名：

```
[me@linuxbox ~]$ ll
```

总结

在这一章中，我们学到了用文本编辑器来编辑配置文件的必要技巧。随着继续学习，当我们 读到命令的手册页时，记录下命令所支持的环境变量。可能会有一个或两个宝贝。在随后的章节 里面，我们将会学习 shell 函数，一个很强大的特性，你可以把它包含在 bash 启动文件里面，以此 来添加你自定制的命令宝库。

拓展阅读

bash 手册页的 INVOCATION 部分非常详细地讨论了 bash 启动文件。

vi 简介

有一个古老的笑话，说是一个在纽约的游客向行人打听这座城市中著名古典音乐场馆的方向：

游客： 请问一下，我怎样去卡内基音乐大厅？

行人： 练习，练习，练习!

学习 Linux 命令行，就像要成为一名造诣很深的钢琴家一样，它不是我们一下午就能学会的技能。这需要 经历几年的勤苦练习。在这一章中，我们将介绍 vi（发音“vee eye”）文本编辑器，它是 Unix 传统中核心程序之一。 vi 因它难用的用户界面而有点声名狼藉，但是当我们看到一位大师坐在钢琴前开始演奏时，我们的确成了 伟大艺术的见证人。虽然我们在这里不能成为 vi 大师，但是当我们学完这一章后， 我们会知道怎样在 vi 中玩“筷子”。

为什么我们应该学习 vi

在现在这个图形编辑器和易于使用的基于文本编辑器的时代，比如说 nano，为什么我们还应该学习 vi 呢？ 下面有三个充分的理由：

- vi 很多系统都预装。如果我们的系统没有图形界面，比方说一台远端服务器或者是一个 X 配置损坏了的本地系统，那么 vi 就成了我们的救星。虽然 nano 逐渐流行起来，但是它 还没有普及。POSIX，这套 Unix 系统中程序兼容的标准，就要求系统要预装 vi。

- vi 是轻量级且执行快速的编辑器。对于许多任务来说，启动 vi 比起在菜单中找到一个图形化文本编辑器， 再等待编辑器数倍兆字节的数据加载而言，要容易的多。另外，vi 是为了加快输入速度而设计的。 我们将会看到，当一名熟练的 vi 用户在编辑文件时，他或她的手从不需要移开键盘。

- 我们不希望其他 Linux 和 Unix 用户把我们看作胆小鬼。

好吧，可能只有两个充分的理由。

一点儿背景介绍

第一版 vi 是在1976由 Bill Joy 写成的，当时他是加州大学伯克利分校的学生， 后来他共同创建了 Sun 微系统公司。vi 这个名字 来源于单词“visual”，因为它打算在带有可移动光标的视频终端上编辑文本。在发明可视化编辑器之前， 有一次只能操作一行文本的行编辑器。为了指定一个修改，我们告诉行编辑器到一个特殊行并且 说明做什么修改，比方说添加或删除文本。视频终端（而不是基于打印机的终端，像电传打印机）的出现 ，可视化编辑成为可能。vi 实际上整合了一个强大的叫做 ex 行编辑器, 所以我们在使用 vi 时能运行行编辑命令。

大多数 Linux 发行版不包含真正的 vi；而是自带一款高级替代版本，叫做 vim（它是“vi improved”的简写）由 Bram Moolenaar 开发的。vim 相对于传统的 Unix vi 来说，取得了实质性进步。通常，vim 在 Linux 系统中是“vi”的符号链接（或别名）。 在随后的讨论中，我们将会假定我们有一个叫做“vi”的程序，但它其实是 vim。

启动和停止 vi

要想启动 vi，只要简单地输入以下命令：

```
[me@linuxbox ~]$ vi
```

一个像这样的屏幕应该出现：

```
VIM - Vi Improved
....
```

正如我们之前操作 nano 时，首先要学的是怎样退出 vi。要退出 vi，输入下面的命令（注意冒号是命令的一部分）：

```
:q
```

shell 提示符应该返回。如果由于某种原因，vi 不能退出（通常因为我们对文件做了修改，却没有保存文件）。 通过给命令加上叹号，我们可以告诉 vi 我们真要退出 vi。

```
:q!
```

小贴示：如果你在 vi 中“迷失”了，试着按下 Esc 键两次来找到路（回到普通模式）。

> 兼容模式
>
> 上面实例中的启动屏幕（来自于 Ubuntu 8.04）， 我们看到一行文字“以 Vi 兼容的模式运行”。这意味着 vim 将以近似于 vi 常规的模式 运行，而不是 vim 的高级规范。为了这章的目的，我们想要使用 vim 的高级规范。要想这样做， 你有几个选择：
>
> 用 vim 来代替 vi。
>
> 如果命令生效，考虑在你的.bashrc 文件中添加别名 vi='vim'。
>
> 或者，使用这个命令在你的 vim 配置文件中添加一行：
>
> echo "set nocp" >> ~/.vimrc
>
> 不同的 Linux 发行版其 vim 软件包也迥然不同。一些发行版只是安装了 vim 的最小版本， 其默认只支持有限的 vim 特性。当练习随后的课程时，你可能会遇到缺失的功能。 如果是这种情况，就安装 vim 的完整版。

编辑模式

再次启动 vi，这次传递给 vi 一个不存在的文件名。这也是用 vi 创建新文件的方法。

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ vi foo.txt
```

如果一切运行正常，我们应该获得一个像这样的屏幕：

```
....
"foo.txt" [New File]
```

每行开头的波浪号（"~"）指示那一行不存在文本。这表示我们有一个空文件。还没有输入任何字符？

学习 vi 时，要知道的第二件非常重要的事情是（知道了如何退出 vi 后）vi 是一个模式编辑器。当 vi 启动后，进入 的是命令模式。这种模式下，几乎每个按键都是一个命令，所以如果我们打算输入字符，vi 会发疯，弄得一团糟。

插入模式

为了在文件中添加文本，首先我们必须进入插入模式。按下"i"按键进入插入模式。之后，我们应该 在屏幕底部看到下面一行，如果 vi 运行在高级模式下（这不会出现在 vi 兼容模式下）：

```
-- INSERT --
```

现在我们能输入一些文本了。试着输入这些文本：

```
The quick brown fox jumped over the lazy dog.
```

按下 Esc 按键，退出插入模式并返回命令模式。

保存我们的工作

为了保存我们刚才对文件所做的修改，我们必须在命令模式下输入一个 ex 命令。 通过按下":"键，这很容易完成。按下冒号键之后，一个冒号字符应该出现在屏幕的底部：

```
:
```

为了写入我们修改的文件，我们在冒号之后输入"w"字符，然后按下回车键：

```
:w
```

文件将会写入到硬盘，并且我们应该在屏幕底部得到一个确认信息，就像这样：

```
"foo.txt" [New] 1L, 46C written
```

小贴示：如果你阅读 vim 的文档，你注意到（混淆地）命令模式被叫做普通模式，ex 命令 叫做命令模式。当心。

移动光标

当在 vi 命令模式下时，vi 提供了大量的移动命令，其中一些是与 less 阅读器共享的。这里 列举了一些：

表13-1: 光标移动按键

按键	移动光标
l or 右箭头	向右移动一个字符
h or 左箭头	向左移动一个字符
j or 下箭头	向下移动一行
k or 上箭头	向上移动一行
0 (零按键)	移动到当前行的行首。
^	移动到当前行的第一个非空字符。
$	移动到当前行的末尾。
w	移动到下一个单词或标点符号的开头。
W	移动到下一个单词的开头，忽略标点符号。
b	移动到上一个单词或标点符号的开头。
B	移动到上一个单词的开头，忽略标点符号。
Ctrl-f or Page Down	向下翻一页
Ctrl-b or Page Up	向上翻一页
numberG	移动到第 number 行。例如，1G 移动到文件的第一行。
G	移动到文件末尾。

为什么 h，j，k，和 l 按键被用来移动光标呢？因为在开发 vi 之初，并不是所有的视频终端都有 箭头按键，熟练的打字员可以使用规则的键盘按键来移动光标，他们的手从不需要移开键盘。

vi 中的许多命令都可以在前面加上一个数字，比方说上面提到的"G"命令。在命令之前加上一个 数字，我们就可以指定命令执行的次数。例如，命令"5j"导致 vi 向下移动5行。

基本编辑

大多数编辑工作由一些基本的操作组成，比如说插入文本，删除文本和通过剪切和粘贴来移动文本。 vi，当然，以它自己的独特方式来支持所有的操作。vi 也提供了有限的撤销形式。如果我们按下“u” 按键，当在命令模式下，vi 将会撤销你所做的最后一次修改。当我们试着执行一些基本的 编辑命令时，这会很方便。

追加文本

vi 有几种不同进入插入模式的方法。我们已经使用了 i 命令来插入文本。

让我们返回到我们的 foo.txt 文件中，呆一会儿：

```
The quick brown fox jumped over the lazy dog.
```

如果我们想要在这个句子的末尾添加一些文本，我们会发现 i 命令不能完成任务，因为我们不能把 光标移到行尾。vi 提供了追加文本的命令，明智地命名为"a"命令。如果我们把光标移动到行尾，输入"a", 光标就会越过行尾，vi 进入插入模式。这样就允许我们添加更多的文本：

```
The quick brown fox jumped over the lazy dog. It was cool.
```

记住按下 Esc 按键来退出插入模式。

因为我们几乎总是想要在行尾附加文本，所以 vi 提供了一种快捷方式来移动到当前行的末尾，并且能添加 文本。它是"A"命令。试着用一下它，给文件添加更多行。

首先，使用"0"(零)命令，将光标移动到行首。现在我们输入"A"，来添加以下文本行：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

再一次，按下 Esc 按键退出插入模式。

正如我们所看到的，大 A 命令非常有用，因为在启动插入模式之前，它把光标移动了行尾。

打开一行

我们插入文本的另一种方式是“打开”一行。这会在存在的两行之间插入一个空白行，并且进入插入模式。 这种方式有两个变体：

表13-2: 文本行打开按键

命令	打开行
o	当前行的下方打开一行。
O	当前行的上方打开一行。

我们可以演示一下：把光标放到"Line 3"上，按下小 o 按键。

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3

line 4
line 5
```

在第三行之下打开了新的一行，并且进入插入模式。按下 Esc，退出插入模式。按下 u 按键，撤销我们的修改。

按下大 O 按键在光标之上打开新的一行：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2

Line 3
Line 4
Line 5
```

按下 Esc 按键，退出插入模式，并且按下 u 按键，撤销我们的更改。

删除文本

正如我们期望的，vi 提供了各种各样的方式来删除文本，所有的方式包含一个或两个按键。首先， x 按键会删除光标位置的一个字符。可以在 x 命令之前带上一个数字，来指明要删除的字符个数。 d 按键更通用一些。类似 x 命令，d 命令之前可以带上一个数字，来指定要执行的删除次数。另外， d 命令之后总是带上一个移动命令，用来控制删除的范围。这里有些实例：

表13-3: 文本删除命令

命令	删除的文本
x	当前字符
3x	当前字符及其后的两个字符。
dd	当前行。
5dd	当前行及随后的四行文本。
dW	从光标位置开始到下一个单词的开头。
d$	从光标位置开始到当前行的行尾。
d0	从光标位置开始到当前行的行首。
d^	从光标位置开始到文本行的第一个非空字符。
dG	从当前行到文件的末尾。
d20G	从当前行到文件的第20行。

把光标放到第一行单词“It”之上。重复按下 x 按键直到删除剩下的部分。下一步，重复按下 u 按键 直到恢复原貌。

注意：真正的 vi 只是支持单层面的 undo 命令。vim 则支持多个层面的。

我们再次执行删除命令，这次使用 d 命令。还是移动光标到单词"It"之上，按下的 dW 来删除单词：

```
The quick brown fox jumped over the lazy dog. was cool.
Line 2
Line 3
Line 4
Line 5
```

按下 d$删除从光标位置到行尾的文本：

```
The quick brown fox jumped over the lazy dog.
Line 2
Line 3
Line 4
Line 5
```

按下 dG 按键删除从当前行到文件末尾的所有行：

```
~
....
```

连续按下 u 按键三次，来恢复删除部分。

剪切，复制和粘贴文本

这个 d 命令不仅删除文本，它还“剪切”文本。每次我们使用 d 命令，删除的部分被复制到一个 粘贴缓冲区中（看作剪切板）。过后我们执行小 p 命令把剪切板中的文本粘贴到光标位置之后， 或者是大 P 命令把文本粘贴到光标之前。

y 命令用来“拉”（复制）文本，和 d 命令剪切文本的方式差不多。这里有些把 y 命令和各种移动命令 结合起来使用的实例：

表13-4: 复制命令

命令	复制的内容
yy	当前行。
5yy	当前行及随后的四行文本。
yW	从当前光标位置到下一个单词的开头。
y$	从当前光标位置到当前行的末尾。
y0	从当前光标位置到行首。
y^	从当前光标位置到文本行的第一个非空字符。
yG	从当前行到文件末尾。
y20G	从当前行到文件的第20行。

我们试着做些复制和粘贴工作。把光标放到文本第一行，输入 yy 来复制当前行。下一步，把光标移动 最后一行（G），输入小写的 p 把复制的一行粘贴到当前行的下面：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog. It was cool.
```

和以前一样，u 命令会撤销我们的修改。光标仍然位于文件的最后一行，输入大写的 P 命令把 所复制的文本粘贴到当前行之上：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog. It was cool.
```

试着执行上表中一些其他的 y 命令，了解小写 p 和大写 P 命令的行为。当你完成练习之后，把文件 恢复原样。

连接行

vi 对于行的概念相当严格。通常，不可能把光标移到行尾，再删除行尾结束符（回车符）来连接 当前行和它下面的一行。由于这个原因，vi 提供了一个特定的命令，大写的 J（不要与小写的 j 混淆了， j 是用来移动光标的）把行与行之间连接起来。

如果我们把光标放到 line 3上，输入大写的 J 命令，看看发生什么情况：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3 Line 4
Line 5
```

查找和替换

vi 有能力把光标移到搜索到的匹配项上。vi 可以在单一行或整个文件中运用这个功能。 它也可以在有或没有用户确认的情况下实现文本替换。

查找一行

f 命令查找一行，移动光标到下一个所指定的字符上。例如，命令 fa 会把光标定位到同一行中 下一个出现的"a"字符上。在一行中执行了字符的查找命令之后，通过输入分号来重复这个查找。

查找整个文件

移动光标到下一个出现的单词或短语上，使用/命令。这个命令和我们之前在 less 程序中学到 的一样。当你输入/命令后，一个"/"字符会出现在屏幕底部。下一步，输入要查找的单词或短语后， 按下回车。光标就会移动到下一个包含所查找字符串的位置。通过 n 命令来重复先前的查找。 这里有个例子：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

把光标移动到文件的第一行。输入：

```
/Line
```

然后键入回车。光标会移动到第二行。下一步， 输入 n，光标移到第三行。重复这个 n 命令，光标会 继续向下移动直到遍历了所有的匹配项。虽然目前，我们只是使用了单词和短语来作为我们的查找 模式，但是 vi 允许使用正则表达式，一种强大的用来表示复杂文本模式的方法。我们将会在随后 的章节里面详尽地介绍正则表达式。

全局查找和替代

vi 使用 ex 命令来执行查找和替代操作（vi 中叫做“替换”）。把整个文件中的单词“Line”更改为“line”， 我们输入以下命令：

```
:%s/Line/line/g
```

我们把这个命令分解为几个单独的部分，看一下每部分的含义：

条目	含义
:	冒号字符运行一个 ex 命令。
	指定要操作的行数。% 是一个快捷方式，表示从第一行到最后一行。另外，操作范围也

%	可以用 1,5 来代替（因为我们的文件只有5行文本），或者用 1,$ 来代替，意思是 “ 从第一行到文件的最后一行。” 如果省略了文本行的范围，那么操作只对当前行生效。
s	指定操作。在这种情况下是，替换（查找与替代）。
/Line/line	查找类型与替代文本。
g	这是“全局”的意思，意味着对文本行中所有匹配的字符串执行查找和替换操作。如果省略 g，则 只替换每个文本行中第一个匹配的字符串。

执行完查找和替代命令之后，我们的文件看起来像这样：

```
The quick brown fox jumped over the lazy dog. It was cool.
line 2
line 3
line 4
line5
```

我们也可以指定一个需要用户确认的替换命令。通过添加一个"c"字符到这个命令的末尾，来完成 这个替换命令。例如：

```
:%s/line/Line/gc
```

这个命令会把我们的文件恢复先前的模样；然而，在执行每个替换命令之前，vi 会停下来， 通过下面的信息，来要求我们确认这个替换：

```
replace with Line (y/n/a/q/l/^E/^Y)?
```

括号中的每个字符都是一个可能的选择，如下所示：

表13-5: 替换确认按键

按键	行为
y	执行替换操作
n	跳过这个匹配的实例
a	对这个及随后所有匹配的字符串执行替换操作。
q or esc	退出替换操作。
l	执行这次替换并退出。l 是 “last” 的简写。
Ctrl-e, Ctrl-y	分别是向下滚动和向上滚动。用于查看建议替换的上下文。

如果你输入 y，则执行这个替换，输入 n 则会导致 vi 跳过这个实例，而移到下一个匹配项上。

编辑多个文件

同时能够编辑多个文件是很有用的。你可能需要更改多个文件或者从一个文件复制内容到 另一个文件。通过 vi，我们可以打开多个文件来编辑，只要在命令行中指定要编辑的文件名。

```
vi file1 file2 file3...
```

我们先退出已经存在的 vi 会话，然后创建一个新文件来编辑。输入:wq 来退出 vi 并且保存了所做的修改。 下一步，我们将在主目录下创建一个额外的用来玩耍的文件。通过获取从 ls 命令的输出，来创建这个文件。

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

用 vi 来编辑我们的原文件和新创建的文件：

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

vi 启动，我们会看到第一个文件显示出来：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

文件之间转换

从这个文件转到下一个文件，使用这个 ex 命令：

```
:n
```

回到先前的文件使用：

```
:N
```

当我们从一个文件移到另一个文件时，如果当前文件没有保存修改，vi 会阻止我们转换文件， 这是 vi 强制执行的政策。在命令之后添加感叹号，可以强迫 vi 放弃修改而转换文件。

另外，上面所描述的转换方法，vim（和一些版本的 vi）也提供了一些 ex 命令，这些命令使 多个文件更容易管理。我们可以查看正在编辑的文件列表，使用:buffers 命令。运行这个 命令后，屏幕顶部就会显示出一个文件列表：

```
:buffers
1 #     "foo.txt"               line 1
2 %a    "ls-output.txt"         line 0
Press ENTER or type command to continue
```

注意：你不同通过:n 或:N 命令在由:e 命令加载的文件之间进行切换。这时要使用:buffer 命令， 其后加上缓冲区号码，来转换文件。

从一个文件复制内容到另一个文件

当我们编辑多个文件时，经常地要复制文件的一部分到另一个正在编辑的文件。使用之前我们学到的 拉（yank）和粘贴命令，这很容易完成。说明如下。以打开的两个文件为例，首先转换到缓冲区1（foo.txt），输入：

```
:buffer 1
```

我们应该得到以下输出：

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

下一步，把光标移到第一行，并且输入 yy 来复制这一行。

转换到第二个缓冲区，输入：

```
:buffer 2
```

现在屏幕会包含一些文件列表（这里只列出了一部分）：

```
total 343700
-rwxr-xr-x 1 root root    31316  2007-12-05  08:58 [
....
```

移动光标到第一行，输入 p 命令把我们从前面文件中复制的一行粘贴到这个文件中：

```
total 343700
The quick brown fox jumped over the lazy dog. It was cool.
-rwxr-xr-x 1 root root    31316  2007-12-05  08:58 [
....
```

插入整个文件到另一个文件

也有可能把整个文件插入到我们所编辑的文件中。看一下实际操作，结束 vi 会话，重新 启动一个只打开一个文件的 vi 会话：

```
[me@linuxbox ~]$ vi ls-output.txt
```

再一次看到我们的文件列表：

```
total 343700
-rwxr-xr-x 1 root root    31316  2007-12-05  08:58 [
```

移动光标到第三行，然后输入以下 ex 命令：

```
:r foo.txt
```

这个:r 命令（是"read"的简称）把指定的文件插入到光标位置之前。现在屏幕应该看起来像这样：

```
total 343700
-rwxr-xr-x 1 root root     31316 2007-12-05  08:58 [
....
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
-rwxr-xr-x 1 root root     111276 2008-01-31  13:36 a2p
....
```

保存工作

像 vi 中的其它操作一样，有几种不同的方法来保存我们所修改的文件。我们已经研究了:w 这个 ex 命令， 但还有几种方法，可能我们也觉得有帮助。

在命令模式下，输入 ZZ 就会保存并退出当前文件。同样地，ex 命令:wq 把:w 和:q 命令结合到 一起，来完成保存和退出任务。

这个:w 命令也可以指定可选的文件名。这个的作用就如"Save As..."。例如，如果我们 正在编辑 foo.txt 文件，想要保存一个副本，叫做 foo1.txt，那么我们可以执行以下命令：

```
:w foo1.txt
```

注意：当上面的命令以一个新名字保存文件时，但它并没有更改你正在编辑的文件的名字。 如果你继续编辑的话，你还是在编辑文件 foo.txt，而不是 foo1.txt。

拓展阅读

即使把这章所学的内容都加起来，我们也只是学了 vi 和 vim 的一点儿皮毛而已。这里 有一些在线的资料，你可以用来继续 vi 学习之旅。

- 学习 vi 编辑器－一本来自于 Wikipedia 的 Wikibook，是一本关于 vi 的简要指南，并 介绍了几个类似 vi 的程序，其中包括 vim。它可以在以下链接中得到：

  http://en.wikibooks.org/wiki/Vi

- The Vim Book－vim 项目，一本570页的书籍，包含了（几乎）所有的 vim 特性。你能在下面链接中找到它：
- Wikipedia 上关于 Bill Joy 的文章，vi 的创始人。

  http://en.wikipedia.org/wiki/Bill_Joy

- Wikipedia 上关于 Bram Moolenaar 的文章，vim 的作者：

  http://en.wikipedia.org/wiki/Bram_Moolenaar

自定制 shell 提示符

在这一章中，我们将会看一下表面上看来很琐碎的细节－shell 提示符。但这会揭示一些内部 shell 和 终端仿真器的工作方式。

和 Linux 内的许多程序一样，shell 提示符是可高度配置的，虽然我们把它相当多地看作是理所当然的， 但是我们一旦学会了怎样控制它，shell 提示符是一个真正有用的设备。

解剖一个提示符

我们默认的提示符看起来像这样：

```
[me@linuxbox ~]$
```

注意它包含我们的用户名，主机名和当前工作目录，但是它又是怎样得到这些东西的呢？结果证明非常简单。提示符是由一个环境变量定义的，叫做 PS1（是“prompt string one” 的简写）。我们可以通过 echo 命令来查看 PS1的内容。

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

注意：如果你 shell 提示符的内容和上例不是一模一样，也不必担心。每个 Linux 发行版 定义的提示符稍微有点不同，其中一些相当异乎寻常。

从输出结果中，我们看到那个 PS1 环境变量包含一些这样的字符，比方说中括号，@符号，和美元符号， 但是剩余部分就是个谜。我们中一些机敏的人会把这些看作是由反斜杠转义的特殊字符，就像我们 在第八章中看到的一样。这里是一部分字符列表，在提示符中 shell 会特殊对待这些字符：

表14－1：Shell 提示符中用到的转义字符

序列	显示值
\a	以 ASCII 格式编码的铃声 . 当遇到这个转义序列时，计算机会发出嗡嗡的响声。
\d	以日，月，天格式来表示当前日期。例如，“Mon May 26.”
\h	本地机的主机名，但不带末尾的域名。
\H	完整的主机名。
\j	运行在当前 shell 会话中的工作数。
\l	当前终端设备名。
\n	一个换行符。
\r	一个回车符。
\s	shell 程序名。
\t	以24小时制，hours:minutes:seconds 的格式表示当前时间.
\T	以12小时制表示当前时间。
\@	以12小时制，AM/PM 格式来表示当前时间。
\A	以24小时制，hours:minutes 格式表示当前时间。
\u	当前用户名。
\v	shell 程序的版本号。
\V	Version and release numbers of the shell.

\w	当前工作目录名。
\W	当前工作目录名的最后部分。
!	当前命令的历史号。
#	当前 shell 会话中的命令数。
$	这会显示一个 '$' 字符，除非你拥有超级用户权限。在那种情况下， 它会显示一个 '#' 字符。
[	标志着一系列一个或多个非打印字符的开始。这被用来嵌入非打印 的控制字符，这些字符以某种方式来操作终端仿真器，比方说移动光标或者是更改文本颜色。
]	标志着非打印字符序列结束。

试试一些可替代的提示符设计

参照这个特殊字符列表，我们可以更改提示符来看一下效果。首先， 我们把原来提示符字符串的内容备份一下，以备之后恢复原貌。为了完成备份， 我们把已有的字符串复制到另一个 shell 变量中，这个变量是我们自己创造的。

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

我们新创建了一个叫做 ps1_old 的变量，并把变量 PS1的值赋 ps1_old。通过 echo 命令可以证明 我们的确复制了 PS1的值。

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

在终端会话中，我们能在任一时间复原提示符，只要简单地反向操作就可以了。

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

现在，我们准备开始，让我们看看如果有一个空的字符串会发生什么：

```
[me@linuxbox ~]$ PS1=
```

如果我们没有给提示字符串赋值，那么我们什么也得不到。根本没有提示字符串！提示符仍然在那里， 但是什么也不显示，正如我们所要求的那样。我们将用一个最小的提示符来代替它：

```
PS1="\$ "
```

这样要好一些。至少能看到我们在做什么。注意双引号中末尾的空格。当提示符显示的时候， 这个空格把美元符号和光标分离开。

在提示符中添加一个响铃：

```
$ PS1="\a\$ "
```

现在每次提示符显示的时候，我们应该能听到嗡嗡声。这会变得很烦人，但是它可能会 很有用，特别是当一个需要运行很长时间的命令执行完后，我们要得到通知。

下一步，让我们试着创建一个信息丰富的提示符，包含主机名和当天时间的信息。

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

试试其他上表中列出的转义序列，看看你能否想出精彩的新提示符。

添加颜色

大多数终端仿真器程序支持一定的非打印字符序列来控制，比方说字符属性（像颜色，黑体和可怕的闪烁） 和光标位置。我们会更深入地讨论光标位置，但首先我们要看一下字体颜色。

> 混乱的终端时代
>
> 回溯到终端连接到远端计算机的时代，有许多竞争的终端品牌，它们各自工作不同。 它们有着不同的键盘，以不同的方式来解释控制信息。Unix 和类似于 Unix 的系统有两个 相当复杂的子系统来处理终端控制领域的混乱局面（称为 termcap 和 terminfo）。如果你 查看一下终端仿真器最底层的属性设置，可能会找到一个关于终端仿真器类型的设置。
>
> 为了努力使所有的终端都讲某种通用语言，美国国家标准委员会（ANSI）制定了 一套标准的字符序列集合来控制视频终端。原先 DOS 用户会记得 ANSI.SYS 文件， 这是一个用来使这些编码解释生效的文件。

字符颜色是由发送到终端仿真器的一个嵌入到了要显示的字符流中的 ANSI 转义编码来控制的。 这个控制编码不会“打印”到屏幕上，而是被终端解释为一个指令。正如我们在上表看到的字符序列， 这个 [和] 序列被用来封装这些非打印字符。一个 ANSI 转义编码以一个八进制033（这个编码是由 退出按键产生的）开头，其后跟着一个可选的字符属性，在之后是一个指令。例如，把文本颜色 设为正常（attribute = 0），黑色文本的编码如下：

```
\033[0;30m
```

这里是一个可用的文本颜色列表。注意这些颜色被分为两组，由应用程序粗体字符属性（1） 分化开来，这个属性可以描绘出“浅”色文本。

表14-2: 用转义序列来设置文本颜色

序列	文本颜色	序列	文本颜色
\033[0;30m	黑色	\033[1;30m	深灰色
\033[0;31m	红色	\033[1;31m	浅红色
\033[0;32m	绿色	\033[1;32m	浅绿色
\033[0;33m	棕色	\033[1;33m	黄色
\033[0;34m	蓝色	\033[1;34m	浅蓝色
\033[0;35m	粉红	\033[1;35m	浅粉色
\033[0;36m	青色	\033[1;36m	浅青色
\033[0;37m	浅灰色	\033[1;37m	白色

让我们试着制作一个红色提示符。我们将在开头加入转义编码：

```
<me@linuxbox ~>$ PS1='\[\033[0;31m\]<\u@\h \W>\$'
<me@linuxbox ~>$
```

我们的提示符生效了，但是注意我们在提示符之后输入的文本也是红色的。为了修改这个问题， 我们将添加另一个转义编码到这个提示符的末尾来告诉终端仿真器恢复到原来的颜色。

```
<me@linuxbox ~>$ PS1='\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\]'
<me@linuxbox ~>$
```

这看起来要好些！

也有可能要设置文本的背景颜色，使用下面列出的转义编码。这个背景颜色不支持黑体属性。

表14-3: 用转义序列来设置背景颜色

\033[0;40m	蓝色	\033[1;44m	黑色
\033[0;41m	红色	\033[1;45m	粉红
\033[0;42m	绿色	\033[1;46m	青色
\033[0;43m	棕色	\033[1;47m	浅灰色

我们可以创建一个带有红色背景的提示符，只是对第一个转义编码做个简单的修改。

```
<me@linuxbox ~>$ PS1='\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] '
<me@linuxbox ~>$
```

试试这些颜色编码，看看你能定制出怎样的提示符！

注意：除了正常的 (0) 和黑体 (1) 字符属性之外，文本也可以具有下划线 (4)，闪烁 (5)， 和反向 (7) 属性。为了拥有好品味，然而，许多终端仿真器拒绝使用这个闪烁属性。

移动光标

转义编码也可以用来定位光标。这些编码被普遍地用来，每次当提示符出现的时候，会在屏幕的不同位置 比如说上面一个角落，显示一个时钟或者其它一些信息。这里是一系列用来定位光标的转义编码：

表14－4： 光标移动转义序列

转义编码	行动
\033[l;cH	把光标移到第 l 行，第 c 列。
\033[nA	把光标向上移动 n 行。
\033[nB	把光标向下移动 n 行。
\033[nC	把光标向前移动 n 个字符。
\033[nD	把光标向后移动 n 个字符。
\033[2J	清空屏幕，把光标移到左上角（第零行，第零列）。
\033[K	清空从光标位置到当前行末的内容。
\033[s	存储当前光标位置。
\033[u	唤醒之前存储的光标位置。

使用上面的编码，我们将构建一个提示符，每次当这个提示符出现的时候，会在屏幕的上方画出一个 包含时钟（由黄色文本渲染）的红色长条。提示符的编码就是这个看起来令人敬畏的字符串：

```
PS1='\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ '
```

让我们分别看一下这个字符串的每一部分所表示的意思：

序列	行动
[	开始一个非打印字符序列。其真正的目的是为了让 bash 能够正确地计算提示符的大小。如果没有这个转义字符的话，命令行编辑 功能会弄错光标的位置。
\033[s	存储光标位置。这个用来使光标能回到原来提示符的位置， 当长条和时钟显示到屏幕上方之后。当心一些 终端仿真器不推崇这个编码。
\033[0;0H	把光标移到屏幕左上角，也就是第零行，第零列的位置。
\033[0;41m	把背景设置为红色。
\033[K	清空从当前光标位置到行末的内容。因为现在 背景颜色是红色，则被清空行背景成为红色，以此来创建长条。注意虽然一直清空到行末， 但是不改变光标位置，它仍然在屏幕左上角。
\033[1;33m	把文本颜色设为黄色。
\t	显示当前时间。虽然这是一个可“打印”的元素，但我们仍把它包含在提示符的非打印部分， 因为我们不想 bash 在计算可见提示符的真正大小时包括这个时钟在内。
\033[0m	关闭颜色设置。这对文本和背景都起作用。
\033[u	恢复到之前保存过的光标位置处。
]	结束非打印字符序列。
<\u@\h \W>$	提示符字符串。

保存提示符

显然地，我们不想总是敲入那个怪物，所以我们将要把这个提示符存储在某个地方。通过把它 添加到我们的.bashrc 文件，可以使这个提示符永久存在。为了达到目的，把下面这两行添加到.bashrc 文件中。

```
PS1='\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ '
export PS1
```

总结归纳

不管你信不信，还有许多事情可以由提示符来完成，涉及到我们在这里没有论及的 shell 函数和脚本， 但这是一个好的开始。并不是每个人都会花心思来更改提示符，因为通常默认的提示符就很让人满意。 但是对于我们这些喜欢思考的人们来说，shell 却提供了许多制造琐碎乐趣的机会。

拓展阅读

- The Bash Prompt HOWTO 来自于 Linux 文档工程，对 shell 提示符的用途进行了相当 完备的论述。可在以下链接中得到：

  http://tldp.org/HOWTO/Bash-Prompt-HOWTO/

- Wikipedia 上有一篇关于 ANSI Escape Codes 的好文章：

  http://en.wikipedia.org/wiki/ANSI_escape_code

软件包管理

如果我们花些时间在 Linux 社区里，我们会得知很多针对, 类如在众多 Linux 发行版中哪个是最好的(等问题的)看法。 这些集中在像这些事情上的讨论，比方说最漂亮的桌面背景（一些人不使用 Ubuntu， 只是因为 Ubuntu 默认主题颜色是棕色的！）和其它的琐碎东西，经常变得非常无聊。

Linux 发行版本质量最重要的决定因素是软件包管理系统和其支持社区的持久性。随着我们 花更多的时间在 Linux 上，我们会发现它的软件园地是非常动态的。软件不断变化。大多数一线 Linux 发行版每隔六个月发布一个新版本，并且许多独立的程序每天都会更新。为了能和这些 如暴风雪一般多的软件保持联系，我们需要一些好工具来进行软件包管理。

软件包管理是指系统中一种安装和维护软件的方法。今天，通过从 Linux 发行版中安装的软件包， 已能满足许多人所有需要的软件。这不同于早期的 Linux，人们需要下载和编辑源码来安装软件。 编辑源码没有任何问题，事实上，拥有对源码的访问权限是 Linux 的伟大奇迹。它赋予我们（ 其它每个人）才干来检测和提高系统性能。只是若有一个预先编译好的软件包处理起来要相对 容易快速些。这章中，我们将查看一些用于包管理的命令行工具。虽然所有主流 Linux 发行版都 提供了强大且精致的图形管理程序来维护系统，但是学习命令行程序也非常重要。因为它们 可以完成许多让图形化管理程序处理起来困难（或者不可能）的任务。

打包系统

不同的 Linux 发行版使用不同的打包系统，一般而言，大多数发行版分别属于两大包管理技术阵营： Debian 的".deb"，和红帽的".rpm"。也有一些重要的例外，比方说 Gentoo， Slackware，和 Foresight，但大多数会使用这两个基本系统中的一个。

表15-1: 主要的包管理系统家族

包管理系统	发行版 (部分列表)
Debian Style (.deb)	Debian, Ubuntu, Xandros, Linspire
Red Hat Style (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE, Mandriva, PCLinuxOS

软件包管理系统是怎样工作的

在专有软件产业中找到的软件发布方法通常需要买一张安装媒介，比方说"安装盘"，然后运行 "安装向导"，来在系统中安装新的应用程序。

Linux 不是这样。Linux 系统中几乎所有的软件都可以在互联网上找到。其中大多数软件由发行商以 包文件的形式提供，剩下的则以源码形式存在，可以手动安装。在后面章节里，我们将会谈谈怎样 通过编译源码来安装软件。

包文件

在包管理系统中软件的基本单元是包文件。包文件是一个构成软件包的文件压缩集合。一个软件包 可能由大量程序以及支持这些程序的数据文件组成。除了安装文件之外，软件包文件也包括 关于这个包的元数据，如软件包及其内容的文本说明。另外，许多软件包还包括预安装和安装后脚本， 这些脚本用来在软件安装之前和之后执行配置任务。

软件包文件是由软件包维护者创建的，他通常是（但不总是）一名软件发行商的雇员。软件维护者 从上游提供商（程序作者）那里得到软件源码，然后编辑源码，创建软件包元数据以及所需要的 安装脚本。通常，软件包维护者要把所做的修改应用到最初的源码当中，来提高此软件与 Linux 发行版其它部分的融合性。

资源库

虽然某些软件项目选择执行他们自己的打包和发布策略，但是现在大多数软件包是由发行商和感兴趣 的第三方创建的。系统发行版的用户可以在一个中心资源库中得到这些软件包，这个资源库可能 包含了成千上万个软件包，每一个软件包都是专门为这个系统发行版建立和维护的。

因软件开发生命周期不同阶段的需要，一个系统发行版可能维护着几个不同的资源库。例如，通常会 有一个"测试"资源库，其中包含刚刚建立的软件包，它们想要勇敢的用户来使用， 在这些软件包正式发布之前，让用户查找错误。系统发行版经常会有一个"开发"资源库， 这个资源库中保存着注定要包含到下一个主要版本中的半成品软件包。

一个系统发行版可能也会拥有相关第三方的资源库。这些资源库需要支持一些因法律原因， 比如说专利或者是 DRM 反规避问题，而不能被包含到发行版中的软件。可能最著名的案例就是 那个加密的 DVD 支持，在美国这是不合法的。第三方资源库在这些软件专利和反规避法案不 生效的国家中起作用。这些资源库通常完全地独立于它们所支持的资源库，要想使用它们， 你必须了解它们，手动地把它们包含到软件包管理系统的配置文件中。

依赖性

程序很少是"孤立的"，而是依赖于其它软件组件来完成它们的工作。常见活动，以 输入/输出为例，就是由共享程序例程来处理的。这些程序例程存储在共享库中，共享库不只 为一个程序提供基本服务。如果一个软件包需要共享资源，比如说共享库，据说就有一个依赖。 现代的软件包管理系统都提供了一些依赖项解析方法，以此来确保当安装软件包时，也安装了 其所有的依赖程序。

上层和底层软件包工具

软件包管理系统通常由两种工具类型组成：底层工具用来处理这些任务，比方说安装和删除软件包文件， 和上层工具，完成元数据搜索和依赖解析。在这一章中，我们将看一下由 Debian 风格的系统 （比如说 Ubuntu，还有许多其它系统）提供的工具，还有那些由 Red Hat 产品使用的工具。虽然所有基于 Red Hat 风格的发行版都依赖于相同的底层程序（rpm）, 但是它们却使用不同的上层工具。我们将研究上层程序 yum 供我们讨论，Fedora, Red Hat 企业版，和 CentOs 都是使用 yum。其它基于 Red Hat 风格的发行版提供了带有可比较特性的上层工具。

表15-2: 包管理工具

发行版	底层工具	上层工具
Debian-Style	dpkg	apt-get, aptitude
Fedora, Red Hat Enterprise Linux, CentOS	rpm	yum

常见软件包管理任务

通过命令行软件包管理工具可以完成许多操作。我们将会看一下最常用的工具。注意底层工具也 支持软件包文件的创建，这个话题超出了本书叙述的范围。在以下的讨论中，"package_name" 这个术语是指软件包实际名称，而不是指"package_file"，它是包含在软件包中的文件名。

查找资源库中的软件包

使用上层工具来搜索资源库元数据，可以根据软件包的名字和说明来定位它。

表15-3: 软件包查找工具

风格	命令
Debian	apt-get update apt-cache search search_string
Red Hat	yum search search_string

例如：搜索一个 yum 资源库来查找 emacs 文本编辑器，使用以下命令：

```
yum search emacs
```

从资源库中安装一个软件包

上层工具允许从一个资源库中下载一个软件包，并经过完全依赖解析来安装它。

表15-4: 软件包安装命令

风格	命令
Debian	apt-get update apt-get install package_name

Red Hat	yum install package_name

例如：从一个 apt 资源库来安装 emacs 文本编辑器：

```
apt-get update; apt-get install emacs
```

通过软件包文件来安装软件

如果从某处而不是从资源库中下载了一个软件包文件，可以使用底层工具来直接（没有经过依赖解析）安装它。

表15-5: 底层软件包安装命令

风格	命令
Debian	dpkg --install package_file
Red Hat	rpm -i package_file

例如：如果已经从一个并非资源库的网站下载了软件包文件 emacs-22.1-7.fc7-i386.rpm， 则可以通过这种方法来安装它：

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

注意：因为这项技术使用底层的 rpm 程序来执行安装任务，所以没有运行依赖解析。 如果 rpm 程序发现缺少了一个依赖，则会报错并退出。

卸载软件

可以使用上层或者底层工具来卸载软件。下面是可用的上层工具。

表15-6: 软件包删除命令

风格	命令
Debian	apt-get remove package_name
Red Hat	yum erase package_name

例如：从 Debian 风格的系统中卸载 emacs 软件包：

```
apt-get remove emacs
```

经过资源库来更新软件包

最常见的软件包管理任务是保持系统中的软件包都是最新的。上层工具仅需一步就能完成 这个至关重要的任务。

表15-7: 软件包更新命令

风格	命令
Debian	apt-get update; apt-get upgrade
Red Hat	yum update

例如：更新安装在 Debian 风格系统中的软件包：

```
apt-get update; apt-get upgrade
```

经过软件包文件来升级软件

如果已经从一个非资源库网站下载了一个软件包的最新版本，可以安装这个版本，用它来 替代先前的版本：

表15-8: 底层软件包升级命令

风格	命令
Debian	dpkg --install package_file
Red Hat	rpm -U package_file

例如：把 Red Hat 系统中所安装的 emacs 的版本更新到软件包文件 emacs-22.1-7.fc7-i386.rpmz 所包含的 emacs 版本。

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

注意：dpkg 程序与安装软件相比没有一个特定的选项，如 rpm 程序那样，来升级一个软件包，。

列出所安装的软件包

下表中的命令可以用来显示安装到系统中的所有软件包列表：

表15-9: 列出所安装的软件包命令

风格	命令
Debian	dpkg --list
Red Hat	rpm -qa

确定是否安装了一个软件包

这些底端工具可以用来显示是否安装了一个指定的软件包：

表15-10: 软件包状态命令

风格	命令
Debian	dpkg --status package_name
Red Hat	rpm -q package_name

例如：确定是否 Debian 风格的系统中安装了这个 emacs 软件包：

```
dpkg --status emacs
```

显示所安装软件包的信息

如果知道了所安装软件包的名字，使用以下命令可以显示这个软件包的说明信息：

表15-11: 查看软件包信息命令

风格	命令
Debian	apt-cache show package_name
Red Hat	yum info package_name

例如：查看 Debian 风格的系统中 emacs 软件包的说明信息：

```
apt-cache show emacs
```

查找安装了某个文件的软件包

确定哪个软件包对所安装的某个特殊文件负责，使用下表中的命令：

表15-12: 包文件识别命令

风格	命令
Debian	dpkg --search file_name
Red Hat	rpm -qf file_name

例如：在 Red Hat 系统中，查看哪个软件包安装了/usr/bin/vim 这个文件

```
rpm -qf /usr/bin/vim
```

总结归纳

在随后的章节里面，我们将探讨许多不同的程序，这些程序涵盖了广泛的应用程序领域。虽然 大多数程序一般是默认安装的，但是若所需程序没有安装在系统中，那么我们可能需要安装额外的软件包。 通过我们新学到的（和了解的）软件包管理知识，我们应该没有问题来安装和管理所需的程序。

> *Linux 软件安装谣言*
>
> 从其它平台迁移过来的用户有时会成为谣言的受害者，说是在 Linux 系统中，安装软件有些 困难，并且不同系统发行版所使用的各种各样的打包方案是一个障碍。唉，它是一个障碍， 但只是针对于那些希望把他们的秘密软件只以二进制版本发行的专有软件供应商。
>
> Linux 软件生态系统是基于开放源代码理念。如果一个程序开发人员发布了一款产品的 源码，那么与系统发行版相关联的开发人员可能就会把这款产品打包，并把它包含在 他们的资源库中。这种方法保证了这款产品能很好地与系统发行版整合在一起，同时为用户 “一站式采购”软件提供了方便，从而用户不必去搜索每个产品的网站。
>
> 设备驱动差不多也以同样的方式来处理，但它们不是系统发行版资源库中单独的项目， 它们本身是 Linux 系统内核的一部分。一般来说，在 Linux 当中没有一个类似于“驱动盘”的东西。 要不内核支持一个设备，要不不支持，反正 Linux 内核支持很多设备，事实上，多于 Windows 所支持的设备数目。当然，如果你需要的特定设备不被支持，这里也没有安慰。当那种情况 发生时，你需要查找一下原因。缺少驱动程序支持通常是由以下三种情况之一导致：
>
> 1. 设备太新。 因为许多硬件供应商没有积极地支持 Linux 的发展，那么编写内核 驱动代码的任务就由一些 Linux 社区来承担，而这需要花费时间。
>
> 2. 设备太奇异。 不是所有的发行版都包含每个可能的设备驱动。每个发行版会建立 它们自己的内核，因为内核是可以配置的（这使得从手表到主机的每台设备上运行 Linux 成为可能）， 这样它们可能会忽略某个特殊设备。通过定位和下载驱动程序的源码，可能需要你自己（是的，由你） 来编译和安装驱动。这个过程不是很难，而是参与。我们将在随后的章节里来讨论编译软件。
>
> 3. 硬件供应商隐藏信息。 他们既不发布应用于 Linux 系统的驱动程序代码， 也不发布技术文档来让某人创建它。这意味着硬件供应商试图保密此设备的程序接口。因为我们 不想在计算机中使用保密的设备，所以我建议删除这令人厌恶的软件， 把它和其它无用的项目都仍到垃圾桶里。

拓展阅读

花些时间来了解你所用发行版中的软件包管理系统。每个发行版都提供了关于自带软件包管理工具的 文档。另外，这里有一些更普遍的资源：

- Debian GNU/Linux FAQ 关于软件包管理一章对软件包管理进行了概述：

  http://www.debian.org/doc/FAQ/ch-pkgtools.en.html

- RPM 工程的主页：

  http://www.rpm.org

- 杜克大学 YUM 工程的主页：

  http://linux.duke.edu/projects/yum/

- 了解一点儿背景知识，Wikipedia 上有一篇关于 metadata 的文章：

  http://en.wikipedia.org/wiki/Metadata

存储媒介

在前面章节中，我们已经从文件级别看了操作数据。在这章里，我们将从设备级别来考虑数据。 Linux 有着令人惊奇的能力来处理存储设备，不管是物理设备，比如说硬盘，还是网络设备，或者是 虚拟存储设备，像 RAID（独立磁盘冗余阵列)和 LVM（逻辑卷管理器）。

然而，这不是一本关于系统管理的书籍，我们不会试图深入地覆盖整个主题。我们将努力做的就是 介绍一些概念和用来管理存储设备的重要命令。

我们将会使用 USB 闪存，CD-RW 光盘（因为系统配备了 CD-ROM 烧写器）和一张软盘（若系统这样配备）， 来做这章的练习题。

我们将看看以下命令：

- mount – 挂载一个文件系统
- umount – 卸载一个文件系统
- fsck – 检查和修复一个文件系统
- fdisk – 分区表控制器
- mkfs – 创建文件系统
- fdformat – 格式化一张软盘
- dd — 把面向块的数据直接写入设备
- genisoimage (mkisofs) – 创建一个 ISO 9660的映像文件
- wodim (cdrecord) – 把数据写入光存储媒介
- md5sum – 计算 MD5检验码

挂载和卸载存储设备

Linux 桌面系统的最新进展已经使存储设备管理对于桌面用户来说极其容易。大多数情况下，我们 只要把设备连接到系统中，它就能工作。在过去（比如说，2004年），这个工作必须手动完成。 在非桌面系统中（例如，服务器中），这仍然是一个主要地手动过程，因为服务器经常有极端的存储需求 和复杂的配置要求。

管理存储设备的第一步是把设备连接到文件系统树中。这个过程叫做挂载，允许设备参与到操作系统中。 回想一下第三章，类似于 Unix 的操作系统，像 Linux，维护单一文件系统树，设备连接到各个结点上。 这与其它操作系统形成对照，比如说 MS-DOS 和 Windows 系统中，每个设备（例如 C:\，D:\，等） 保持着单独的文件系统树。

有一个叫做/etc/fstab 的文件可以列出系统启动时要挂载的设备（典型地，硬盘分区）。下面是 来自于 Fedora 7系统的/etc/fstab 文件实例：

```
LABEL=/12              /               ext3        defaults        1   1
LABEL=/home            /home           ext3        defaults        1   2
LABEL=/boot            /boot           ext3        defaults        1   2
tmpfs                  /dev/shm        tmpfs       defaults        0   0
devpts                 /dev/pts        devpts      gid=5,mode=620  0   0
sysfs                  /sys            sysfs       defaults        0   0
proc                   /proc           proc        defaults        0   0
LABEL=SWAP-sda3        /swap           swap        defaults        0   0
```

在这个实例中所列出的大多数文件系统是虚拟的，并不适用于我们的讨论。就我们的目的而言， 前三个是我们感兴趣的：

```
LABEL=/12              /              ext3       defaults        1   1
LABEL=/home            /home          ext3       defaults        1   2
LABEL=/boot            /boot          ext3       defaults        1   2
```

这些是硬盘分区。每行由六个字段组成，如下所示：

表16－1： /etc/fstab 字段

字段	内容	说明
1	设备名	传统上，这个字段包含与物理设备相关联的设备文件的实际名字，比如说/dev/hda1（第一个 IDE 通道上第一个主设备分区）。然而今天的计算机，有很多热插拔设备（像 USB 驱动设备），许多 现代的 Linux 发行版用一个文本标签和设备相关联。当这个设备连接到系统中时，这个标签（当储存媒介格式化时，这个标签会被添加到存储媒介中）会被操作系统读取。 那样的话，不管赋给实际物理设备哪个设备文件，这个设备仍然能被系统正确地识别。
2	挂载点	设备所连接到的文件系统树的目录。
3	文件系统类型	Linux 允许挂载许多文件系统类型。大多数本地的 Linux 文件系统是 ext3， 但是也支持很多其它的，比方说 FAT16 (msdos), FAT32 (vfat)，NTFS (ntfs)，CD-ROM (iso9660)，等等。
4	选项	文件系统可以通过各种各样的选项来挂载。有可能，例如，挂载只读的文件系统， 或者挂载阻止执行任何程序的文件系统（一个有用的安全特性，避免删除媒介。）
5	频率	一位数字，指定是否和在什么时间用 dump 命令来备份一个文件系统。
6	次序	一位数字，指定 fsck 命令按照什么次序来检查文件系统。

查看挂载的文件系统列表

这个 mount 命令被用来挂载文件系统。执行这个不带参数的命令，将会显示 一系列当前挂载的文件系统：

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

这个列表的格式是：设备 on 挂载点 type 文件系统类型（可选的）。例如，第一行所示设备/dev/sda2 作为根文件系统被挂载，文件系统类型是 ext3，并且可读可写（这个“rw”选项）。在这个列表的底部有 两个有趣的条目。倒数第二行显示了在读卡器中的一张2G 的 SD 内存卡，挂载到了/media/disk 上。最后一行 是一个网络设备，挂载到了/misc/musicbox 上。

第一次实验，我们将使用一张 CD-ROM。首先，在插入 CD-ROW 之前，我们将看一下系统：

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

这个列表来自于 CentOS 5系统，使用 LVM（逻辑卷管理器）来创建它的根文件系统。正如许多现在的 Linux 发行版一样，这个 系统试图自动挂载插入的 CD-ROM。当我们插入光盘后，我们看看下面的输出：

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,
nodev,uid=500)
```

当我们插入光盘后，除了额外的一行之外，我们看到和原来一样的列表。在列表的末尾，我们 看到 CD-ROW 已经挂载到了/media/live-1.0.10-8上，它的文件类型是 iso9660（CD-ROW）。 就我们的实验目的而言，我们对这个设备的名字感兴趣。当你自己进行这个实验时，这个 设备名字是最有可能不同的。

警告：在随后的实例中，至关重要的是你要密切注意用在你系统中的实际设备名，并且 不要使用此文本中使用的名字！

还要注意音频 CD 和 CD-ROW 不一样。音频 CD 不包含文件系统，这样在通常意义上，它就不能被挂载了。

现在我们拥有 CD-ROW 光盘的设备名字，让我们卸载这张光盘，并把它重新挂载到文件系统树 的另一个位置。我们需要超级用户身份（使用系统相应的命令）来进行操作，并且用 umount（注意这个命令的拼写）来卸载光盘：

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/hdc
```

下一步是创建一个新的光盘挂载点。简单地说，一个挂载点就是文件系统树中的一个目录。它没有 什么特殊的。它甚至不必是一个空目录，即使你把设备挂载到了一个非空目录上，你也不能看到 这个目录中原来的内容，直到你卸载这个设备。就我们的目的而言，我们将创建一个新目录：

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

最后，我们把这个 CD-ROW 挂载到一个新的挂载点上。这个-t 选项用来指定文件系统类型：

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

之后，我们可以通过这个新挂载点来查看 CD-ROW 的内容：

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

注意当我们试图卸载这个 CD-ROW 时，发生了什么事情。

```
[root@linuxbox cdrom]# umount /dev/hdc
umount: /mnt/cdrom: device is busy
```

这是怎么回事呢？原因是我们不能卸载一个设备，如果某个用户或进程正在使用这个设备的话。在这种 情况下，我们把工作目录更改到了 CD-ROW 的挂载点，这个挂载点导致设备忙碌。我们可以很容易地修复这个问题 通过把工作目录改到其它目录而不是这个挂载点。

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

现在这个设备成功卸载了。

> 为什么卸载重要
>
> 如果你看一下 free 命令的输出结果，这个命令用来显示关于内存使用情况的统计信息，你 会看到一个统计值叫做"buffers“。计算机系统旨在尽可能快地运行。系统运行速度的 一个阻碍是缓慢的设备。打印机是一个很好的例子。即使最快速的打印机相比于计算机标准也 极其地缓慢。一台计算机确实会运行地非常慢，如果它要停下来等待一台打印机打印完一页。 在早期的个人电脑时代（多任务之前），这真是个问题。如果你正在编辑电子表格 或者是文本文档，每次你要打印文件时，计算机都会停下来而且变得不能使用。 计算机能以打印机可接受的最快速度把数据发送给打印机，但由于打印机不能快速地打印， 这个发送速度会非常慢。这个问题被解决了，由于打印机缓存的出现，一个包含一些 RAM 内存 的设备，位于计算机和打印机之间。通过打印机缓存，计算机把要打印的结果发送到这个缓存区， 数据会迅速地存储到这个 RAM 中，这样计算机就能回去工作，而不用等待。与此同时，打印机缓存将会 以打印机可接受的速度把缓存中的数据缓慢地输出给打印机。
>
> 缓存被广泛地应用于计算机中，使其运行地更快。别让偶尔地需要读取或写入慢设备阻碍了 系统的运行速度。在实际与慢设备交互之前，操作系统会尽可能多的读取或写入数据到内存中的 存储设备里。以 Linux 操作系统为例，你会注意到系统看似填充了多于它所需要的内存。 这不意味着 Linux 正在使用所有的内存，它意味着 Linux 正在利用所有可用的内存，来作为缓存区。
>
> 这个缓存区允许非常快速地写入存储设备，因为写入物理设备的操作被延迟到后面进行。同时， 这些注定要传送到设备中的数据正在内存中堆积起来。时不时地，操作系统会把这些数据 写入物理设备。
>
> 卸载一个设备需要把所有剩余的数据写入这个设备，所以设备可以被安全地移除。如果 没有卸载设备，就移除了它，就有可能没有把注定要发送到设备中的数据输送完毕。在某些情况下， 这些数据可能包含重要的目录更新信息，这将导致文件系统损坏，这是发生在计算机中的最坏的事情之一

确定设备名称

有时很难来确定设备名称。在以前，这并不是很难。一台设备总是在某个固定的位置，也不会 挪动它。类似于 Unix 的系统喜欢设备那样安排。之前在开发 Unix 系统的时候，“更改一个磁盘驱动器”要用一辆 叉车从机房中移除一台如洗衣机大小的设备。最近几年，典型的桌面硬件配置已经变得相当动态，并且 Linux 已经发展地比其祖先更加灵活。在以上事例中，我们利用现代 Linux 桌面系统的功能来“自动地”挂载 设备，然后再确定设备名称。但是如果我们正在管理一台服务器或者是其它一些（这种自动挂载功能）不会 发生的环境，我们又如何能查清设备名呢？

首先，让我们看一下系统怎样来命名设备。如果我们列出目录/dev（所有设备的住所）的内容，我们 会看到许许多多的设备：

```
[me@linuxbox ~]$ ls /dev
```

这个列表的内容揭示了一些设备命名的模式。这里有几个：

表16－2：Linux 存储设备名称

模式	设备
/dev/fd	软盘驱动器
/dev/hd	老系统中的 IDE(PATA)磁盘。典型的主板包含两个 IDE 连接器或者是通道，每个连接器 带有一根缆线，每根缆线上有两个硬盘驱动器连接点。缆线上的第一个驱动器叫做主设备， 第二个叫做从设备。设备名称这样安排，/dev/hdb 是指第一通道上的主设备名；/dev/hdb 是第一通道上的从设备名；/dev/hdc 是第二通道上的主设备名，等等。末尾的数字表示 硬盘驱动器上的分区。例如，/dev/hda1是指系统中第一硬盘驱动器上的第一个分区，而 /dev/hda 则是指整个硬盘驱动器。
/dev/lp	打印机
/dev/sd	SCSI 磁盘。在最近的 Linux 系统中，内核把所有类似于磁盘的设备（包括 PATA/SATA 硬盘， 闪存，和 USB 存储设备，比如说可移动的音乐播放器和数码相机）看作 SCSI 磁盘。 剩下的命名系统类似于上述所描述的旧的/dev/hd命名方案。
/dev/sr	光盘（CD/DVD 读取器和烧写器）

另外，我们经常看到符号链接比如说/dev/cdrom，/dev/dvd 和/dev/floppy，它们指向实际的 设备文件，提供这些链接是为了方便使用。如果你工作的系统不能自动挂载可移动的设备，你可以使用 下面的技巧来决定当可移动设备连接后，它是怎样被命名的。首先，启动一个实时查看文件/var/log/messages （你可能需要超级用户权限）：

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

这个文件的最后几行会被显示，然后停止。下一步，插入这个可移动的设备。在 这个例子里，我们将使用一个16MB 闪存。瞬间，内核就会发现这个设备， 并且探测它：

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device
using uhci_hcd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen
from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass
Storage devices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short
(5), using 36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy
Disk 1.00 PQ: 0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic
sg3 type 0
```

显示再次停止之后，输入 Ctrl-c，重新得到提示符。输出结果的有趣部分是一再提及“[sdb]”， 这正好符和我们期望的 SCSI 磁盘设备名称。知道这一点后，有两行输出变得颇具启发性：

```
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
```

这告诉我们这个设备名称是/dev/sdb 指整个设备，/dev/sdb1是这个设备的第一分区。 正如我们所看到的，使用 Linux 系统充满了有趣的监测工作。

小贴士：使用这个 tail -f /var/log/messages 技巧是一个很不错的方法，可以实时 观察系统的一举一动。

既然知道了设备名称，我们就可以挂载这个闪存驱动器了：

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
Filesystem      1K-blocks   Used       Available   Use%    Mounted on
/dev/sda2       15115452    5186944     9775164     35%     /
/dev/sda5       59631908    31777376    24776480    57%     /home
/dev/sda1       147764      17277       122858      13%     /boot
tmpfs           776808      0           776808      0%      /dev/shm
/dev/sdb1       15560       0           15560       0%      /mnt/flash
```

这个设备名称会保持不变只要设备与计算机保持连接并且计算机不会重新启动。

创建新的文件系统

假若我们想要用 Linux 本地文件系统来重新格式化这个闪存驱动器，而不是它现用的 FAT32系统。 这涉及到两个步骤：1.（可选的）创建一个新的分区布局若已存在的分区不是我们喜欢的。2. 在这个闪存上创建一个新的空的文件系统。

注意！在下面的练习中，我们将要格式化一个闪存驱动器。拿一个不包含有用数据的驱动器 作为实验品，因为它将会被擦除！再次，请确定你指定了正确的系统设备名称。未能注意此 警告可能导致你格式化（即擦除）错误的驱动器！

用 **fdisk** 命令操作分区

这个 fdisk 程序允许我们直接在底层与类似磁盘的设备（比如说硬盘驱动器和闪存驱动器）进行交互。 使用这个工具可以在设备上编辑，删除，和创建分区。以我们的闪存驱动器为例， 首先我们必须卸载它（如果需要的话），然后调用 fdisk 程序，如下所示：

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

注意我们必须指定设备名称，就整个设备而言，而不是通过分区号。这个程序启动后，我们 将看到以下提示：

```
Command (m for help):
```

输入"m"会显示程序菜单：

```
Command action
a       toggle a bootable flag
....
```

我们想要做的第一件事情是检查已存在的分区布局。输入"p"会打印出这个设备的分区表：

```
Command (m for help): p

Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

Device Boot     Start       End     Blocks   Id       System
/dev/sdb1           2       1008      15608+   b       w95 FAT32
```

在此例中，我们看到一个16MB 的设备只有一个分区(1)，此分区占用了可用的1008个柱面中的1006个, 并被标识为 Windows 95 FAT32分区。有些程序会使用这个标志符来限制一些可以对磁盘所做的操作， 但大多数情况下更改这个标志符没有危害。然而，为了叙述方便，我们将会更改它， 以此来表明是个 Linux 分区。在更改之前，首先我们必须找到被用来识别一个 Linux 分区的 ID 号码。 在上面列表中，我们看到 ID 号码“b”被用来指定这个已存在的分区。要查看可用的分区类型列表， 参考之前的程序菜单。我们会看到以下选项：

```
l   list known partition types
```

如果我们在提示符下输入“l”，就会显示一个很长的可能类型列表。在它们之中会看到“b”为已存在分区 类型的 ID 号，而“83”是针对 Linux 系统的 ID 号。

回到之前的菜单，看到这个选项来更改分区 ID 号：

```
t   change a partition's system id
```

我们先输入“t”，再输入新的 ID 号：

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

这就完成了我们需要做得所有修改。到目前为止，还没有接触这个设备（所有修改都存储在内存中， 而不是在此物理设备中），所以我们将会把修改过的分区表写入此设备，再退出。为此，我们输入 在提示符下输入"w":

```
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

如果我们已经决定保持设备不变，可在提示符下输入"q"，这将退出程序而没有写更改。我们 可以安全地忽略这些不祥的警告信息。

用 **mkfs** 命令创建一个新的文件系统

完成了分区编辑工作（它或许是轻量级的），是时候在我们的闪存驱动器上创建一个新的文件系统了。 为此，我们会使用 mkfs（"make file system"的简写），它能创建各种格式的文件系统。 在此设备上创建一个 ext3文件系统，我们使用"-t" 选项来指定这个"ext3"系统类型，随后是我们要格式化的设备分区名称：

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2007)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
8193
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

当 ext3被选为文件系统类型时，这个程序会显示许多信息。若把这个设备重新格式化为它最初的 FAT32文件 系统，指定"vfat"作为文件系统类型：

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

任何时候添加额外的存储设备到系统中时，都可以使用这个分区和格式化的过程。虽然我们 只以一个小小的闪存驱动器为例，同样的操作可以被应用到内部硬盘和其它可移动的存储设备上 像 USB 硬盘驱动器。

测试和修复文件系统

在之前讨论文件/etc/fstab 时，我们会在每行的末尾看到一些神秘的数字。每次系统启动时， 在挂载系统之前，都会按照惯例

检查文件系统的完整性。这个任务由 fsck 程序（是"file system check"的简写）完成。每个 fstab 项中的最后一个数字指定了设备的检查顺序。 在上面的实例中，我们看到首先检查根文件系统，然后是 home 和 boot 文件系统。若最后一个数字 是零则相应设备不会被检查。

除了检查文件系统的完整性之外，fsck 还能修复受损的文件系统，其成功度依赖于损坏的数量。 在类似于 Unix 的文件系统中，文件恢复的部分被放置于 lost+found 目录里面，位于每个文件 系统的根目录下面。

检查我们的闪存驱动器（首先应该卸载），我们能执行下面的操作：

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2008)
e2fsck 1.40.8 (13-Mar-2008)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

以我的经验，文件系统损坏情况相当罕见，除非硬件存在问题，如磁盘驱动器故障。 在大多数系统中，系统启动阶段若探测到文件系统已经损坏了，则会导致系统停止下来， 在系统继续执行之前，会指导你运行 fsck 程序。

> 什么是 fsck?
>
> 在 Unix 文化中，"fsck"这个单词往往会被用来代替一个流行的词，“fsck”和这个词共享了三个 字母。这个尤其适用，因为你可能会说出上文提到的词，若你发现自己处于这种境况下， 被强制来运行 fsck 命令时。 {: .single}

格式化软盘

对于那些还在使用配备了软盘驱动器的计算机的用户，我们也能管理这些设备。准备一 张可用的空白软盘要分两个步骤。首先，对这张软盘执行低级格式化，然后创建一个文件系统。 为了完成格式化，我们使用 fdformat 程序，同时指定软盘设备名称（通常为/dev/fd0）：

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

接下来，通过 mkfs 命令，给这个软盘创建一个 FAT 文件系统：

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

注意我们使用这个“msdos”文件系统类型来得到旧（小的）风格的文件分配表。当一个软磁盘 被准备好之后，则可能像其它设备一样挂载它。

直接把数据移入/出设备

虽然我们通常认为计算机中的数据以文件形式来组织数据，也可以“原始的”形式来考虑数据。 如果我们看一下磁盘驱动器，例如， 我们看到它由大量的数据“块”组成，而操作系统却把这些数据块看作目录和文件。然而，如果 把磁盘驱动器简单地看成一个数据块大集合，我们就能执行有用的任务，如克隆设备。

这个 dd 程序能执行此任务。它可以把数据块从一个地方复制到另一个地方。它使用独特的语法（由于历史原因），经常它被这样使用：

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

比方说我们有两个相同容量的 USB 闪存驱动器，并且要精确地把第一个驱动器（中的内容） 复制给第二个。如果连接两个设备到计算机上，它们各自被分配到设备/dev/sdb 和 /dev/sdc 上，这样我们就能通过下面的命令把第一个驱动器中的所有数据复制到第二个 驱动器中。

```
dd if=/dev/sdb of=/dev/sdc
```

或者，如果只有第一个驱动器被连接到计算机上，我们可以把它的内容复制到一个普通文件中供 以后恢复或复制数据：

```
dd if=/dev/sdb of=flash_drive.img
```

警告！这个 dd 命令非常强大。虽然它的名字来自于“数据定义”，有时候也把它叫做“清除磁盘” 因为用户经常会误输入 if 或 of 的规范。在按下回车键之前，要再三检查输入与输出规范！

创建 CD-ROM 映像

写入一个可记录的 CD-ROM（一个 CD-R 或者是 CD-RW）由两步组成；首先， 构建一个 iso 映像文件， 这就是一个 CD-ROM 的文件系统映像，第二步，把这个映像文件写入到 CD-ROM 媒介中。

创建一个 CD-ROM 的映像拷贝

如果想要制作一张现有 CD-ROM 的 iso 映像，我们可以使用 dd 命令来读取 CD-ROW 中的所有数据块， 并把它们复制到本地文件中。比如说我们有一张 Ubuntu CD，用它来制作一个 iso 文件，以后我们可以用它来制作更多的拷贝。插入这张 CD 之后，确定 它的设备名称（假定是/dev/cdrom），然后像这样来制作 iso 文件：

```
dd if=/dev/cdrom of=ubuntu.iso
```

这项技术也适用于 DVD 光盘，但是不能用于音频 CD，因为它们不使用文件系统来存储数据。 对于音频 CD，看一下 cdrdao 命令。

从文件集合中创建一个映像

创建一个包含目录内容的 iso 映像文件，我们使用 genisoimage 程序。为此，我们首先创建 一个目录，这个目录中包含了要包括到此映像中的所有文件，然后执行这个 genisoimage 命令 来创建映像文件。例如，如果我们已经创建一个叫做~/cd-rom-files 的目录，然后用文件 填充此目录，再通过下面的命令来创建一个叫做 cd-rom.iso 映像文件：

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

"-R"选项添加元数据为 Rock Ridge 扩展，这允许使用长文件名和 POSIX 风格的文件权限。 同样地，这个"-J"选项使 Joliet 扩展生效，这样 Windows 中就支持长文件名了。

> 如果你看一下关于创建和烧写光介质如 CD-ROMs 和 DVD 的在线文档，你会经常碰到两个程序 叫做 mkisofs 和 cdrecord。这些程序是流行软件包"cdrtools"的一部分，"cdrtools"由 Jorg Schilling 编写成。在2006年春天，Schilling 先生更改了部分 cdrtools 软件包的协议，许多 Linux 社区的意见是， 这创建了一个与 GNU GPL 不相兼容的协议。结果，就 fork 了这个 cdrtools 项目， 目前新项目里面包含 cdrecord 和 mkisofs 的替代程序，分别是 wodim 和 genisoimage。

写入 CD-ROM 镜像

有了一个映像文件之后，我们可以把它烧写到光盘中。下面讨论的大多数命令对可 记录的 CD-ROW 和 DVD 媒介都适用。

直接挂载一个 ISO 镜像

有一个诀窍，我们可以用它来挂载 iso 映像文件，虽然此文件仍然在我们的硬盘中，但我们 当作它已经在光盘中了。添加 "-o loop" 选项来挂载（同时带有必需的 "-t iso9660" 文件系统类型）， 挂载这个映像文件就好像它是一台设备，把它连接到文件

系统树上：

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

上面的示例中，我们创建了一个挂载点叫做/mnt/iso_image，然后把此映像文件 image.iso 挂载到挂载点上。映像文件被挂载之后，可以把它当作，就好像它是一张 真正的 CD-ROM 或者 DVD。当不再需要此映像文件后，记得卸载它。

清除一张可重写入的 **CD-ROM**

可重写入的 CD-RW 媒介在被重使用之前需要擦除或清空。为此，我们可以用 wodim 命令，指定 设备名称和清空的类型。此 wodim 程序提供了几种清空类型。最小（且最快）的是 "fast" 类型：

```
wodim dev=/dev/cdrw blank=fast
```

写入镜像

写入一个映像文件，我们再次使用 wodim 命令，指定光盘设备名称和映像文件名：

```
wodim dev=/dev/cdrw image.iso
```

除了设备名称和映像文件之外，wodim 命令还支持非常多的选项。常见的两个选项是，"-v" 可详细输出， 和 "－dao" 以 disk-at-once 模式写入光盘。如果你正在准备一张光盘为的是商业复制，那么应该使用这种模式。 wodim 命令的默认模式是 track-at-once，这对于录制音乐很有用。

拓展阅读

我们刚才谈到了很多方法，可以使用命令行管理存储介质。看看我们所讲过命令的手册页。 一些命令支持大量的选项和操作。此外，寻找一些如何添加硬盘驱动器到 Linux 系统（有许多）的在线教程， 这些教程也要适用于光介质存储设备。

友情提示

通常验证一下我们已经下载的 iso 映像文件的完整性很有用处。在大多数情况下，iso 映像文件的贡献者也会提供 一个 checksum 文件。一个 checksum 是一个神奇的数学运算的计算结果，这个数学计算会产生一个能表示目标文件内容的数字。 如果目标文件的内容即使更改一个二进制位，checksum 的结果将会非常不一样。 生成 checksum 数字的最常见方法是使用 md5sum 程序。当你使用 md5sum 程序的时候， 它会产生一个独一无二的十六进制数字：

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece    image.iso
```

当你下载完映像文件之后，你应该对映像文件执行 md5sum 命令，然后把运行结果与发行商提供的 md5sum 数值作比较。

除了检查下载文件的完整性之外，我们也可以使用 md5sum 程序验证新写入的光学存储介质。 为此，首先我们计算映像文件的 checksum 数值，然后计算此光学存储介质的 checksum 数值。 这种验证光学介质的技巧是限定只对 光学存储介质中包含映像文件的部分计算 checksum 数值。 通过确定映像文件所包含的 2048 个字节块的数目（光学存储介质总是以 2048 个字节块的方式写入） 并从存储介质中读取那么多的字节块，我们就可以完成操作。 某些类型的存储介质，并不需要这样做。一个以 disk-at-once 模式写入的 CD-R，可以用下面的方式检验：

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece    /dev/cdrom
```

许多存储介质类型，如 DVD 需要精确地计算字节块的数目。在下面的例子中，我们检验了映像文件 dvd-image.iso 以及

DVD 光驱中磁盘 /dev/dvd 文件的完整性。你能弄明白这是怎么回事吗？

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s" dvd-image.iso) / 2048 )) | md5sum
```

网络系统

当谈及到网络系统层面，几乎任何东西都能由 Linux 来实现。Linux 被用来创建各式各样的网络系统和装置， 包括防火墙，路由器，名称服务器，网络连接式存储设备等等。

被用来配置和操作网络系统的命令数目，就如网络系统一样巨大。我们仅仅会关注一些最经常 使用到的命令。我们要研究的命令包括那些被用来监测网络和传输文件的命令。另外，我们 还会探讨用来远端登录的 ssh 程序。这章会介绍：

- ping - 发送 ICMP ECHO_REQUEST 软件包到网络主机
- traceroute - 打印到一台网络主机的路由数据包
- netstat - 打印网络连接，路由表，接口统计数据，伪装连接，和多路广播成员
- ftp - 因特网文件传输程序
- wget - 非交互式网络下载器
- ssh - OpenSSH SSH 客户端（远程登录程序）

我们假定你已经知道了一点网络系统背景知识。在这个因特网时代，每个计算机用户需要理解基本的网络 系统概念。为了能够充分利用这一章节的内容，我们应该熟悉以下术语：

- IP (网络协议)地址
- 主机和域名
- URI（统一资源标识符）

请查看下面的“拓展阅读”部分，有几篇关于这些术语的有用文章。

注意：一些将要讲到的命令可能（取决于系统发行版）需要从系统发行版的仓库中安装额外的软件包， 并且一些命令可能需要超级用户权限才能执行。

检查和监测网络

即使你不是一名系统管理员，检查一个网络的性能和运作情况也是经常有帮助的。

最基本的网络命令是 ping。这个 ping 命令发送一个特殊的网络数据包，叫做 IMCP ECHO_REQUEST，到 一台指定的主机。大多数接收这个包的网络设备将会回复它，来允许网络连接验证。

注意：大多数网络设备（包括 Linux 主机）都可以被配置为忽略这些数据包。通常，这样做是出于网络安全 原因，部分地遮蔽一台主机免受一个潜在攻击者地侵袭。配置防火墙来阻塞 IMCP 流量也很普遍。

例如，看看我们能否连接到网站 linuxcommand.org（我们最喜欢的网站之一）， 我们可以这样使用 ping 命令：

```
[me@linuxbox ~]$ ping linuxcommand.org
```

一旦启动，ping 命令会持续在特定的时间间隔内（默认是一秒）发送数据包，直到它被中断：

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=1
ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=2
ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=3
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=4
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=5
ttl=43 time=105 ms
...
```

按下组合键 Ctrl-c，中断这个命令之后，ping 打印出运行统计信息。一个正常工作的网络会报告 零个数据包丢失。一个成功执行的“ping”命令会意味着网络的各个部件（网卡，电缆，路由，网关） 都处于正常的工作状态。

这个 traceroute 程序（一些系统使用相似的 tracepath 程序来代替）会显示从本地到指定主机 要经过的所有“跳数”的网络流量列表。例如，看一下到达 slashdot.org 网站，需要经过的路由 器，我们将这样做：

```
[me@linuxbox ~]$ traceroute slashdot.org
```

命令输出看起来像这样：

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte
packets
1 ipcop.localdomain (192.168.1.1) 1.066 ms 1.366 ms 1.720 ms
2 * * *
3 ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9) 14.622
ms 14.885 ms 15.169 ms
4 po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154) 17.634
ms 17.626 ms 17.899 ms
5 po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158) 15.992
ms 15.983 ms 16.256 ms
6 po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5) 22.835
...
```

从输出结果中，我们可以看到连接测试系统到 slashdot.org 网站需要经由16个路由器。对于那些 提供标识信息的路由器，我们能看到它们的主机名，IP 地址和性能数据，这些数据包括三次从本地到 此路由器的往返时间样本。对于那些没有提供标识信息的路由器（由于路由器配置，网络拥塞，防火墙等 方面的原因），我们会看到几个星号，正如行中所示。

netstat 程序被用来检查各种各样的网络设置和统计数据。通过此命令的许多选项，我们 可以看看网络设置中的各种特性。使用“-ie”选项，我们能够查看系统中的网络接口：

```
[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100 RX bytes:153098921 (146.0 MB) TX
        bytes:261035246 (248.9 MB) Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
...
```

在上述实例中，我们看到我们的测试系统有两个网络接口。第一个，叫做 eth0，是 因特网接口，和第二个，叫做 lo，是内部回环网络接口，它是一个虚拟接口，系统用它来 “自言自语”。

当执行日常网络诊断时，要查看的重要信息是每个网络接口第四行开头出现的单词 “UP”，说明这个网络接口已经生效，还要查看第二行中 inet addr 字段出现的有效 IP 地址。对于使用 DHCP（动态主机配置协议）的系统，在 这个字段中的一个有效

IP 地址则证明了 DHCP 工作正常。

使用这个“-r”选项会显示内核的网络路由表。这展示了系统是如何配置网络之间发送数据包的。

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination     Gateway     Genmask         Flags    MSS  Window  irtt Iface

192.168.1.0     *           255.255.255.0   U        0    0          0 eth0
default         192.168.1.1 0.0.0.0         UG       0    0          0 eth0
```

在这个简单的例子里面，我们看到了，位于防火墙之内的局域网中，一台客户端计算机的典型路由表。 第一行显示了目的地 192.168.1.0。IP 地址以零结尾是指网络，而不是个人主机， 所以这个目的地意味着局域网中的任何一台主机。下一个字段，Gateway， 是网关（路由器）的名字或 IP 地址，用它来连接当前的主机和目的地的网络。 若这个字段显示一个星号，则表明不需要网关。

最后一行包含目的地 default。指的是发往任何表上没有列出的目的地网络的流量。 在我们的实例中，我们看到网关被定义为地址 192.168.1.1 的路由器，它应该能 知道怎样来处理目的地流量。

netstat 程序有许多选项，我们仅仅讨论了几个。查看 netstat 命令的手册，可以 得到所有选项的完整列表。

网络中传输文件

网络有什么用处呢？除非我们知道了怎样通过网络来传输文件。有许多程序可以用来在网络中 传送数据。我们先讨论两个命令，随后的章节里再介绍几个命令。

ftp 命令属于真正的“经典”程序之一，它的名字来源于其所使用的协议，就是文件传输协议。 FTP 被广泛地用来从因特网上下载文件。大多数，并不是所有的，网络浏览器都支持 FTP， 你经常可以看到它们的 URI 以协议 ftp://开头。在出现网络浏览器之前，ftp 程序已经存在了。 ftp 程序可用来与 FTP 服务器进行通信，FTP 服务器就是存储文件的计算机，这些文件能够通过 网络下载和上传。

FTP（它的原始形式）并不是安全的，因为它会以明码形式发送帐号的姓名和密码。这就意味着 这些数据没有加密，任何嗅探网络的人都能看到。由于此种原因，几乎因特网中所有 FTP 服务器 都是匿名的。一个匿名服务器能允许任何人使用注册名“anonymous”和无意义的密码登录系统。

在下面的例子中，我们将展示一个典型的会话，从匿名 FTP 服务器，其名字是 fileserver， 的/pub/_images/Ubuntu-8.04的目录下，使用 ftp 程序下载一个 Ubuntu 系统映像文件。

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd\_images/Ubuntu-8.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r-- 1 500 500 733079552 Apr 25 03:53 ubuntu-8.04- desktop-i386.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-8.04-desktop-i386.iso
local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-desktop-
i386.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-8.04-desktop-
i386.iso (733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

这里是对会话期间所输入命令的解释说明：

命令	意思
ftp fileserver	唤醒 ftp 程序，让它连接到 FTP 服务器，fileserver。
anonymous	登录名。输入登录名后，将出现一个密码提示。一些服务器将会接受空密码， 其它一些则会要求一个邮件地址形式的密码。如果是这种情况，试着输入 “user@example.com”。
cd pub/cd_images/Ubuntu-8.04	跳转到远端系统中，要下载文件所在的目录下， 注意在大多数匿名的 FTP 服务器中，支持公共下载的文件都能在目录 pub 下找到
ls	列出远端系统中的目录。
lcd Desktop	跳转到本地系统中的 ~/Desktop 目录下。在实例中，ftp 程序在工作目录 ~ 下被唤醒。 这个命令把工作目录改为 ~/Desktop
get ubuntu-8.04-desktop-i386.iso	告诉远端系统传送文件到本地。因为本地系统的工作目录 已经更改到了 ~/Desktop，所以文件会被下载到此目录。
bye	退出远端服务器，结束 ftp 程序会话。也可以使用命令 quit 和 exit。

在 “ftp>” 提示符下，输入 “help”，会显示所支持命令的列表。使用 ftp 登录到一台 授予了用户足够权限的服务器中，则可以执行很多普通的文件管理任务。虽然很笨拙， 但它真能工作。

ftp 并不是唯一的命令行形式的 FTP 客户端。实际上，还有很多。其中比较好（也更流行的）是 lftp 程序， 由 Alexander Lukyanov 编写完成。虽然 lftp 工作起来与传统的 ftp 程序很相似，但是它带有额外的便捷特性，包括 多协议支持（包括 HTTP），若下载失败会自动地重新下载，后台处理，用 tab 按键来补全路径名，还有很多。

另一个流行的用来下载文件的命令行程序是 wget。若想从网络和 FTP 网站两者上都能下载数据，wget 是很有用处的。 不只能下载单个文件，多个文件，甚至整个网站都能下载。下载 linuxcommand.org 网站的首页， 我们可以这样做：

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51-- http://linuxcommand.org/index.php
        => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

  [ <                      => ]        3,120       --.--K/s

11:02:51 (161.75 MB/s) - 'index.php' saved [3120]
```

这个程序的许多选项允许 wget 递归地下载，在后台下载文件（你退出后仍在下载），能完成未下载 全的文件。这些特性在命令手册，better-than-average 一节中有详尽地说明。

与远程主机安全通信

通过网络来远程操控类似 Unix 的操作系统已经有很多年了。早些年，在因特网普遍推广之前，有 一些受欢迎的程序被用来登录远程主机。它们是 rlogin 和 telnet 程序。然而这些程序，拥有和 ftp 程序 一样的致命缺点；它们以明码形式来传输所有的交流信息（包括登录命令和密码）。这使它们完全不 适合使用在因特网时代。

为了解决这个问题，开发了一款新的协议，叫做 SSH（Secure Shell）。 SSH 解决了这两个基本的和远端主机安全交流的问题。首先，它要认证远端主机是否为它 所知道的那台主机（这样就阻止了所谓的“中间人”的攻击），其次，它加密了本地与远程主机之间 所有的通讯信息。

SSH 由两部分组成。SSH 服务器运行在远端主机上运行，在端口号22上监听将要到来的连接，而 SSH 客户端用在本地系统中，用来和远端服务器通信。

大多数 Linux 发行版自带一个提供 SSH 功能的软件包，叫做 OpenSSH，来自于 BSD 项目。一些发行版 默认包含客户端和服务器端两个软件包（例如，Red Hat）,而另一些（比方说 Ubuntu）则只是提供客户端服务。为了能让系统接受远端的连接，它必须 安装 OpenSSH-server 软件包，配置，运行它，并且（如果系统正在运行，或者是在防火墙之后） 它必须允许在 TCP 端口号上接收网络连接。

小贴示：如果你没有远端系统去连接，但还想试试这些实例，则确认安装了 OpenSSH-server 软件包，则可使用 localhost 作为远端主机的名字。这种情况下，计算机会和它自己创建网络连接。

用来与远端 SSH 服务器相连接的 SSH 客户端程序，顺理成章，叫做 ssh。连接到远端名为 remote-sys 的主机，我们可以这样使用 ssh 客户端程序：

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be
established.
RSA key fingerprint is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

第一次尝试连接，提示信息表明远端主机的真实性不能确立。这是因为客户端程序以前从没有 看到过这个远端主机。为了接受远端主机的身份验证凭据，输入“yes”。一旦建立了连接，会提示 用户输入他或她的密码：

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list
of known hosts.
me@remote-sys's password:
```

成功地输入密码之后，我们会接收到远端系统的 shell 提示符：

```
Last login: Sat Aug 30 13:00:48 2008
[me@remote-sys ~]$
```

远端 shell 会话一直存在，直到用户输入 exit 命令后，则关闭了远程连接。这时候，本地的 shell 会话 恢复，本地 shell 提示符重新出现。

也有可能使用不同的用户名连接到远程系统。例如，如果本地用户“me”，在远端系统中有一个帐号名 “bob”，则用户 me 能够用 bob 帐号登录到远端系统，如下所示：

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Sat Aug 30 13:03:21 2008
[bob@remote-sys ~]$
```

正如之前所讲到的，ssh 验证远端主机的真实性。如果远端主机不能成功地通过验证，则会提示以下信息：

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@
WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
...
```

有两种可能的情形会提示这些信息。第一，某个攻击者企图制造“中间人”袭击。这很少见， 因为每个人都知道 ssh 会针对这种状况发出警告。最有可能的罪魁祸首是远端系统已经改变了；例如，它的操作系统或者是 SSH 服务器重新安装了。然而，为了安全起见，第一个可能性不应该 被轻易否定。当这条消息出现时，总要与远端系统的管理员查对一下。

当确定了这条消息归结为一个良性的原因之后，那么在客户端更正问题就很安全了。 使用文本编辑器（可能是 vim）从文件~/.ssh/known_hosts 中删除废弃的钥匙， 就解决了问题。在上面的例子里，我们看到这样一句话：

```
Offending key in /home/me/.ssh/known_hosts:1
```

这意味着文件 known_hosts 里面某一行包含攻击型的钥匙。从文件中删除这一行，则 ssh 程序 就能够从远端系统接受新的身份验证凭据。

除了能够在远端系统中打开一个 shell 会话，ssh 程序也允许我们在远端系统中执行单个命令。 例如，在名为 remote-sys 的远端主机上，执行 free 命令，并把输出结果显示到本地系统 shell 会话中。

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
          total   used       free     shared buffers cached

Mem:        775536  507184   268352          0  110068 154596

-/+ buffers/cache: 242520  533016
Swap: 0 1572856 0 110068 154596

[me@linuxbox ~]$
```

有可能以更有趣的方式来利用这项技术，比方说下面的例子，我们在远端系统中执行 ls 命令， 并把命令输出重定向到本地系统中的一个文件里面。

```
[me@linuxbox ~]$ ssh remote-sys 'ls \*' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

注意，上面的例子中使用了单引号。这样做是因为我们不想路径名展开操作在本地执行 ；而希望 它在远端系统中被执行。同样地，如果我们想要把输出结果重定向到远端主机的文件中，我们可以 把重定向操作符和文件名都放到单引号里面。

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

> *SSH* 通道
>
> 当你通过 SSH 协议与远端主机建立连接的时候，其中发生的事就是在本地与远端系统之间 创建了一条加密通道。通常，这条通道被用来把在本地系统中输入的命令安全地传输到远端系统， 同样地，再把执行结果安全地发送回来。除了这个基本功能之外，SSH 协议允许大多数 网络流量类型通过这条加密通道来被传送，在本地与远端系统之间创建某种 VPN（虚拟专用网络）。
>
> 可能这个特性的最普遍使用是允许传递 X 窗口系统流量。在运行着 X 服务器（也就是， 能显示 GUI 的机器）的系统中，有可能在远端启动和运行一个 X 客户端程序（一个图形化应用程序）， 而应用程序的显示结果出现在本地。这很容易完成，这里有个例子：假设我们正坐在一台装有 Linux 系统， 叫做 linuxbox 的机器之前，且系统中运行着 X 服务器，现在我们想要在名为 remote-sys 的远端系统中 运行 xload 程序，但是要在我们的本地系统中看到这个程序的图形化输出。我们可以这样做：
>
> ```
> [me@linuxbox ~]$ ssh -X remote-sys
> me@remote-sys's password:
> Last login: Mon Sep 08 13:23:11 2008
> [me@remote-sys ~]$ xload
> ```
>
> 这个 xload 命令在远端执行之后，它的窗口就会出现在本地。在某些系统中，你可能需要 使用“－Y”选项，而不是“－X”选项来完成这个操作。

这个 OpenSSH 软件包也包含两个程序，它们可以利用 SSH 加密通道在网络间复制文件。 第一个，scp（安全复制）被用来复制文件，与熟悉的 cp 程序非常相似。最显著的区别就是 源或者目标路径名要以远端主机的名字，后跟一个冒号字符开头。例如，如果我们想要 从远端系统，remote-sys，的主目录下复制文档 document.txt，到我们本地系统的当前工作目录下， 可以这样操作：

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt
100%        5581        5.5KB/s        00:00
[me@linuxbox ~]$
```

和 ssh 命令一样，如果你所期望的远端主机帐户与你本地系统中的不一致， 则可以把用户名添加到远端主机名的开头。

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

第二个 SSH 文件复制命令是 sftp，正如其名字所示，它是 ftp 程序的安全替代品。sftp 工作起来与我们 之前使用的 ftp 程序很相似；然而，它不用明码形式来传递数据，它使用加密的 SSH 通道。sftp 有一个 重要特性强于传统的 ftp 命令，就是 sftp 不需要远端系统中运行 FTP 服务器。它仅仅要求 SSH 服务器。 这意味着任何一台能用 SSH 客户端连接的远端机器，也可当作类似于 FTP 的服务器来使用。 这里是一个样本会话：

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-
desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB 7.4MB/s 01:35
sftp> bye
```

小贴示：这个 SFTP 协议被许多 Linux 发行版中的图形化文件管理器支持。使用 Nautilus (GNOME), 或者是 Konqueror (KDE)，我们都能在位置栏中输入以 sftp://开头的 URI， 来操作存储在运行着 SSH 服务器的远端系统中的文件。

> *Windows 中的 SSH 客户端*
>
> 比方说你正坐在一台 Windows 机器前面，但是你需要登录到你的 Linux 服务器中，去完成 一些实际的工作，那该怎么办呢？当然是得到一个 Windows 平台下的 SSH 客户端！有很多这样 的工具。最流行的可能就是由 Simon Tatham 和他的团队开发的 PuTTY 了。这个 PuTTY 程序 能够显示一个终端窗口，而且允许 Windows 用户在远端主机中打开一个 SSH（或者 telnet）会话。 这个程序也提供了 scp 和 sftp 程序的类似物。
>
> PuTTY 可在链接 http://www.chiark.greenend.org.uk/~sgtatham/putty/ 处得到。

拓展阅读

- Linux 文档项目提供了 Linux 网络管理指南，可以广泛地（虽然过时了）了解网络管理方面的知识。

  http://tldp.org/LDP/nag2/index.html

- Wikipedia 上包含了许多网络方面的优秀文章。这里有一些基础的：

  http://en.wikipedia.org/wiki/Internet_protocol_address

  http://en.wikipedia.org/wiki/Host_name

  http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

查找文件

因为我们已经浏览了 Linux 系统，所以一件事已经变得非常清楚：一个典型的 Linux 系统包含很多文件！ 这就引发了一个问题，“我们怎样查找东西？”。虽然我们已经知道 Linux 文件系统良好的组织结构，是源自 类似于 Unix 的操作系统代代传承的习俗。但是仅文件数量就会引起可怕的问题。在这一章中，我们将察看 两个用来在系统中查找文件的工具。这些工具是：

- locate – 通过名字来查找文件
- find – 在目录层次结构中搜索文件

我们也将看一个经常与文件搜索命令一起使用的命令，它用来处理搜索到的文件列表：

- xargs – 从标准输入生成和执行命令行

另外，我们将介绍两个命令来协助我们探索：

- touch – 更改文件时间
- stat – 显示文件或文件系统状态

locate – 查找文件的简单方法

这个 locate 程序快速搜索路径名数据库，并且输出每个与给定字符串相匹配的文件名。比如说， 例如，我们想要找到所有名字以“zip”开头的程序。因为我们正在查找程序，可以假定包含 匹配程序的目录以"bin/"结尾。因此，我们试着以这种方式使用 locate 命令，来找到我们的文件：

```
[me@linuxbox ~]$ locate bin/zip
```

locate 命令将会搜索它的路径名数据库，输出任一个包含字符串“bin/zip”的路径名：

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

如果搜索要求没有这么简单，locate 可以结合其它工具，比如说 grep 命令，来设计更加 有趣的搜索：

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

这个 locate 程序已经存在了很多年了，它有几个不同的变体被普遍使用着。在现在 Linux 发行版中发现的两个最常见的变体是 slocate 和 mlocate，但是通常它们被名为 locate 的 符号链接访问。不同版本的 locate 命令拥有重复的选项集合。一些版

本包括正则表达式 匹配（我们会在下一章中讨论）和通配符支持。查看 locate 命令的手册，从而确定安装了 哪个版本的 locate 程序。

> *locate* 数据库来自何方？
>
> 你可能注意到了，在一些发行版中，仅仅在系统安装之后，locate 不能工作， 但是如果你第二天再试一下，它就工作正常了。怎么回事呢？locate 数据库由另一个叫做 updatedb 的程序创建。通常，这个程序作为一个 cron 工作例程周期性运转；也就是说，一个任务 在特定的时间间隔内被 cron 守护进程执行。大多数装有 locate 的系统会每隔一天运行一回 updatedb 程序。因为数据库不能被持续地更新，所以当使用 locate 时，你会发现 目前最新的文件不会出现。为了克服这个问题，有可能手动运行 updatedb 程序， 更改为超级用户身份，在提示符下运行 updatedb 命令。

find – 查找文件的复杂方式

locate 程序只能依据文件名来查找文件，而 find 程序能基于各种各样的属性， 搜索一个给定目录（以及它的子目录），来查找文件。我们将要花费大量的时间学习 find 命令，因为 它有许多有趣的特性，当我们开始在随后的章节里面讨论编程概念的时候，我们将会重复看到这些特性。

find 命令的最简单使用是，搜索一个或多个目录。例如，输出我们的主目录列表。

```
[me@linuxbox ~]$ find ~
```

对于最活跃的用户帐号，这将产生一张很大的列表。因为这张列表被发送到标准输出， 我们可以把这个列表管道到其它的程序中。让我们使用 wc 程序来计算出文件的数量：

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

哇，我们一直很忙！find 命令的美丽所在就是它能够被用来识别符合特定标准的文件。它通过 （有点奇怪）应用选项，测试条件，和操作来完成搜索。我们先看一下测试条件。

Tests

比如说我们想要目录列表。我们可以添加以下测试条件：

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

添加测试条件-type d 限制了只搜索目录。相反地，我们使用这个测试条件来限定搜索普通文件：

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

这里是 find 命令支持的普通文件类型测试条件：

表18-1: find 文件类型

文件类型	描述
b	块设备文件
c	字符设备文件
d	目录
f	普通文件
l	符号链接

我们也可以通过加入一些额外的测试条件，根据文件大小和文件名来搜索：让我们查找所有文件名匹配 通配符模式“*.JPG”和文件大小大于1M 的文件：

```
[me@linuxbox ~]$ find ~ -type f -name "\*.JPG" -size +1M | wc -l
840
```

在这个例子里面，我们加入了 -name 测试条件，后面跟通配符模式。注意，我们把它用双引号引起来， 从而阻止 shell 展开路径名。紧接着，我们加入 -size 测试条件，后跟字符串“+1M”。开头的加号表明 我们正在寻找文件大小大于指定数的文件。若字符串以减号开头，则意味着查找小于指定数的文件。 若没有符号意味着“精确匹配这个数”。结尾字母“M”表明测量单位是兆字节。下面的字符可以 被用来指定测量单位：

Table 18-2: find 大小单位

字符	单位
b	512 个字节块。如果没有指定单位，则这是默认值。
c	字节
w	两个字节的字
k	千字节(1024个字节单位)
M	兆字节(1048576个字节单位)
G	千兆字节(1073741824个字节单位)

find 命令支持大量不同的测试条件。下表是列出了一些常见的测试条件。请注意，在需要数值参数的 情况下，可以应用以上讨论的“+”和"-"符号表示法：

表18-3: find 测试条件

测试条件	描述
-cmin n	匹配的文件和目录的内容或属性最后修改时间正好在 n 分钟之前。 指定少于 n 分钟之前，使用 -n，指定多于 n 分钟之前，使用 +n。
-cnewer file	匹配的文件和目录的内容或属性最后修改时间早于那些文件。
-ctime n	匹配的文件和目录的内容和属性最后修改时间在 n*24小时之前。
-empty	匹配空文件和目录。
-group name	匹配的文件和目录属于一个组。组可以用组名或组 ID 来表示。
-iname pattern	就像-name 测试条件，但是大小写敏感。
-inum n	匹配的文件的 inode 号是 n。这对于找到某个特殊 inode 的所有硬链接很有帮助。
-mmin n	匹配的文件或目录的内容被修改于 n 分钟之前。
-mtime n	匹配的文件或目录的内容被修改于 n*24小时之前。
-name pattern	用指定的通配符模式匹配的文件和目录。
-newer file	匹配的文件和目录的内容早于指定的文件。当编写 shell 脚本，做文件备份时，非常有帮助。 每次你制作一个备份，更新文件（比如说日志），然后使用 find 命令来决定自从上次更新，哪一个文件已经更改了。
-nouser	匹配的文件和目录不属于一个有效用户。这可以用来查找 属于删除帐户的文件或监测攻击行为。
-nogroup	匹配的文件和目录不属于一个有效的组。
-perm mode	匹配的文件和目录的权限已经设置为指定的 mode。mode 可以用 八进制或符号表示法。
-samefile name	相似于-inum 测试条件。匹配和文件 name 享有同样 inode 号的文件。
-size n	匹配的文件大小为 n。
-type c	匹配的文件类型是 c。

-user name	匹配的文件或目录属于某个用户。这个用户可以通过用户名或用户 ID 来表示。

This is not a complete list. The find man page has all the details.

这不是一个完整的列表。find 命令手册有更详细的说明。

操作符

即使拥有了 find 命令提供的所有测试条件，我们还需要一个更好的方式来描述测试条件之间的逻辑关系。例如， 如果我们需要确定是否一个目录中的所有的文件和子目录拥有安全权限，怎么办呢？ 我们可以查找权限不是0600的文件和权限不是0700的目录。幸运地是，find 命令提供了 一种方法来结合测试条件，通过使用逻辑操作符来创建更复杂的逻辑关系。 为了表达上述的测试条件，我们可以这样做：

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d -not -perm 0700 \)
```

呀！这的确看起来很奇怪。这些是什么东西？实际上，这些操作符没有那么复杂，一旦你知道了它们的原理。 这里是操作符列表：

表18-4: find 命令的逻辑操作符

操作符	描述
-and	匹配如果操作符两边的测试条件都是真。可以简写为 -a。 注意若没有使用操作符，则默认使用 -and。
-or	匹配若操作符两边的任一个测试条件为真。可以简写为 -o。
-not	匹配若操作符后面的测试条件是真。可以简写为一个感叹号（!）。
()	把测试条件和操作符组合起来形成更大的表达式。这用来控制逻辑计算的优先级。 默认情况下，find 命令按照从左到右的顺序计算。经常有必要重写默认的求值顺序，以得到期望的结果。 即使没有必要，有时候包括组合起来的字符，对提高命令的可读性是很有帮助的。注意 因为圆括号字符对于 shell 来说有特殊含义，所以在命令行中使用它们的时候，它们必须 用引号引起来，才能作为实参传递给 find 命令。通常反斜杠字符被用来转义圆括号字符。

With this list of operators in hand, let’s deconstruct our find command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an -or operator:

通过这张操作符列表，我们重建 find 命令。从最外层看，我们看到测试条件被分为两组，由一个 -or 操作符分开：

```
( expression 1 ) -or ( expression 2 )
```

这很有意义，因为我们正在搜索具有不同权限集合的文件和目录。如果我们文件和目录两者都查找， 那为什么要用 -or 来代替 -and 呢？因为 find 命令扫描文件和目录时，会计算每一个对象，看看它是否 匹配指定的测试条件。我们想要知道它是具有错误权限的文件还是有错误权限的目录。它不可能同时符合这 两个条件。所以如果展开组合起来的表达式，我们能这样解释它：

```
( file with bad perms ) -or ( directory with bad perms )
```

下一个挑战是怎样来检查“错误权限”，这个怎样做呢？我们不从这个角度做。我们将测试 “不是正确权限”，因为我们知道什么是“正确权限”。对于文件，我们定义正确权限为0600， 目录则为0711。测试具有“不正确”权限的文件表达式为：

```
-type f -and -not -perms 0600
```

对于目录，表达式为：

```
-type d -and -not -perms 0700
```

正如上述操作符列表中提到的，这个-and 操作符能够被安全地删除，因为它是默认使用的操作符。 所以如果我们把这两个表达式连起来，就得到最终的命令：

```
find ~ ( -type f -not -perms 0600 ) -or ( -type d -not -perms 0700 )
```

然而，因为圆括号对于 shell 有特殊含义，我们必须转义它们，来阻止 shell 解释它们。在圆括号字符 之前加上一个反斜杠字符来转义它们。

逻辑操作符的另一个特性要重点理解。比方说我们有两个由逻辑操作符分开的表达式：

```
expr1 -operator expr2
```

在所有情况下，总会执行表达式 expr1；然而由操作符来决定是否执行表达式 expr2。这里 列出了它是怎样工作的：

表18-5: find AND/OR 逻辑

expr1 的结果	操作符	**expr2 is...**
真	-and	总要执行
假	-and	从不执行
真	-or	从不执行
假	-or	总要执行

为什么这会发生呢？这样做是为了提高性能。以 -and 为例，我们知道表达式 expr1 -and expr2 不能为真，如果表达式 expr1 的结果为假，所以没有必要执行 expr2。同样地，如果我们有表达式 expr1 -or expr2，并且表达式 expr1 的结果为真，那么就没有必要执行 expr2，因为我们已经知道 表达式 expr1 -or expr2 为真。好，这样会执行快一些。为什么这个很重要？ 它很重要是因为我们能依靠这种行为来控制怎样来执行操作。我们会很快看到...

预定义的操作

让我们做一些工作吧！从 find 命令得到的结果列表很有用处，但是我们真正想要做的事情是操作列表 中的某些条目。幸运地是，find 命令允许基于搜索结果来执行操作。有许多预定义的操作和几种方式来 应用用户定义的操作。首先，让我们看一下几个预定义的操作：

表18-6: 几个预定义的 find 命令操作

操作	描述
-delete	删除当前匹配的文件。
-ls	对匹配的文件执行等同的 ls -dils 命令。并将结果发送到标准输出。
-print	把匹配文件的全路径名输送到标准输出。如果没有指定其它操作，这是 默认操作。
-quit	一旦找到一个匹配，退出。

和测试条件一样，还有更多的操作。查看 find 命令手册得到更多细节。在第一个例子里， 我们这样做：

```
find ~
```

这个命令输出了我们主目录中包含的每个文件和子目录。它会输出一个列表，因为会默认使用-print 操作 ，如果没有指定其它操作的话。因此我们的命令也可以这样表述：

```
find ~ -print
```

我们可以使用 find 命令来删除符合一定条件的文件。例如，来删除扩展名为“.BAK”（这通常用来指定备份文件） 的文件，我们可以使用这个命令：

```
find ~ -type f -name '*.BAK' -delete
```

在这个例子里面，用户主目录（和它的子目录）下搜索每个以.BAK 结尾的文件名。当找到后，就删除它们。

警告：当使用 -delete 操作时，不用说，你应该格外小心。首先测试一下命令， 用 -print 操作代替 -delete，来确认搜索结果。

在我们继续之前，让我们看一下逻辑运算符是怎样影响操作的。考虑以下命令：

```
find ~ -type f -name '*.BAK' -print
```

正如我们所见到的，这个命令会查找每个文件名以.BAK (-name '*.BAK') 结尾的普通文件 (-type f)， 并把每个匹配文件的相对路径名输出到标准输出 (-print)。然而，此命令按这个方式执行的原因，是 由每个测试和操作之间的逻辑关系决定的。记住，在每个测试和操作之间会默认应用 -and 逻辑运算符。 我们也可以这样表达这个命令，使逻辑关系更容易看出：

```
find ~ -type f -and -name '*.BAK' -and -print
```

当命令被充分表达之后，让我们看看逻辑运算符是如何影响其执行的：

测试／行为	只有...的时候，才被执行
-print	只有 -type f and -name '*.BAK'*为真的时候
-name ‘.BAK’	只有 -type f 为真的时候
-type f	总是被执行，因为它是与-and 关系中的第一个测试／行为。

因为测试和行为之间的逻辑关系决定了哪一个会被执行，我们知道测试和行为的顺序很重要。例如， 如果我们重新安排测试和行为之间的顺序，让 -print 行为是第一个，那么这个命令执行起来会截然不同：

```
find ~ -print -and -type f -and -name '*.BAK'
```

这个版本的命令会打印出每个文件（-print 行为总是为真），然后测试文件类型和指定的文件扩展名。

用户定义的行为

除了预定义的行为之外，我们也可以唤醒随意的命令。传统方式是通过 -exec 行为。这个 行为像这样工作：

```
-exec command {} ;
```

这里的 command 就是指一个命令的名字，{}是当前路径名的符号表示，分号是要求的界定符 表明命令结束。这里是一个使用 -exec 行为的例子，其作用如之前讨论的 -delete 行为：

```
-exec rm '{}' ';'
```

重述一遍，因为花括号和分号对于 shell 有特殊含义，所以它们必须被引起来或被转义。

也有可能交互式地执行一个用户定义的行为。通过使用 -ok 行为来代替 -exec，在执行每个指定的命令之前， 会提示用户：

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me    me 224 2007-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me    me 0 2008-09-19 12:53 /home/me/foo.txt
```

在这个例子里面，我们搜索以字符串“foo”开头的文件名，并且对每个匹配的文件执行 ls -l 命令。 使用-ok 行为，会在 ls 命令执行之前提示用户。

提高效率

当 -exec 行为被使用的时候，若每次找到一个匹配的文件，它会启动一个新的指定命令的实例。 我们可能更愿意把所有的搜索结果结合起来，再运行一个命令的实例。例如，而不是像这样执行命令：

```
ls -l file1
ls -l file2
```

我们更喜欢这样执行命令：

```
ls -l file1 file2
```

这样就导致命令只被执行一次而不是多次。有两种方法可以这样做。传统方式是使用外部命令 xargs， 另一种方法是，使用 find 命令自己的一个新功能。我们先讨论第二种方法。

通过把末尾的分号改为加号，就激活了 find 命令的一个功能，把搜索结果结合为一个参数列表， 然后执行一次所期望的命令。再看一下之前的例子，这个：

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me     me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me     me 0 2008-09-19 12:53 /home/me/foo.txt
```

会执行 ls 命令，每次找到一个匹配的文件。把命令改为：

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me     me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me     me 0 2008-09-19 12:53 /home/me/foo.txt
```

虽然我们得到一样的结果，但是系统只需要执行一次 ls 命令。

xargs

这个 xargs 命令会执行一个有趣的函数。它从标准输入接受输入，并把输入转换为一个特定命令的 参数列表。对于我们的例子，我们可以这样使用它：

```
find ~ -type f -name 'foo\*' -print | xargs ls -l
-rwxr-xr-x 1 me     me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me     me 0 2008-09-19 12:53 /home/me/foo.txt
```

这里我们看到 find 命令的输出被管道到 xargs 命令，反过来，xargs 会为 ls 命令构建 参数列表，然后执行 ls 命令。

注意：当被放置到命令行中的参数个数相当大时，参数个数是有限制的。有可能创建的命令 太长以至于 shell 不能接受。当命令行超过系统支持的最大长度时，xargs 会执行带有最大 参数个数的指定命令，然后重复这个过程直到耗尽标准输入。执行带有 --show--limits 选项 的 xargs 命令，来查看命令行的最大值。

> 处理古怪的文件名
>
> 类似于 Unix 的系统允许在文件名中嵌入空格（甚至换行符）。这就给一些程序，如为其它 程序构建参数列表的 xargs 程序，造成了问题。一个嵌入的空格会被看作是一个界定符，生成的 命令会把每个空格分离的单词解释为单独的参数。为了解决这个问题，find 命令和 xarg 程序 允许可选择的使用一个 null 字符作为参数分隔符。一个 null 字符被定义在 ASCII 码中，由数字 零来表示（相反的，例如，空格字符在 ASCII 码中由数字32表示）。find 命令提供的 -print0 行为， 则会产生由 null 字符分离的输出，并且 xargs 命令有一个 --null 选项，这个选项会接受由 null 字符 分离的输入。这里有一个例子：
>
> find ~ -iname '*.jpg' -print0 | xargs --null ls -l
>
> 使用这项技术，我们可以保证所有文件，甚至那些文件名中包含空格的文件，都能被正确地处理。

返回操练场

到实际使用 find 命令的时候了。我们将会创建一个操练场，来实践一些我们所学到的知识。

首先，让我们创建一个包含许多子目录和文件的操练场：

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

惊叹于命令行的强大功能！只用这两行，我们就创建了一个包含一百个子目录，每个子目录中 包含了26个空文件的操练场。试试用 GUI 来创建它！

我们用来创造这个奇迹的方法中包含一个熟悉的命令（mkdir），一个奇异的 shell 扩展（大括号） 和一个新命令，touch。通过结合 mkdir 命令和-p 选项（导致 mkdir 命令创建指定路径的父目录），以及 大括号展开，我们能够创建一百个目录。

这个 touch 命令通常被用来设置或更新文件的访问，更改，和修改时间。然而，如果一个文件名参数是一个 不存在的文件，则会创建一个空文件。

在我们的操练场中，我们创建了一百个名为 file-A 的文件实例。让我们找到它们：

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

注意不同于 ls 命令，find 命令的输出结果是无序的。其顺序由存储设备的布局决定。为了确定实际上 我们拥有一百个此文件的实例，我们可以用这种方式来确认：

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
```

下一步，让我们看一下基于文件的修改时间来查找文件。当创建备份文件或者以年代顺序来 组织文件的时候，这会很有帮助。为此，首先我们将创建一个参考文件，我们将与其比较修改时间：

```
[me@linuxbox ~]$ touch playground/timestamp
```

这个创建了一个空文件，名为 timestamp，并且把它的修改时间设置为当前时间。我们能够验证 它通过使用另一个方便的命令，stat，是一款加大马力的 ls 命令版本。这个 stat 命令会展示系统对 某个文件及其属性所知道的所有信息：

```
[me@linuxbox ~]$ stat playground/timestamp
File: 'playground/timestamp'
Size: 0 Blocks: 0 IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me) Gid: ( 1001/ me)
Access: 2008-10-08 15:15:39.000000000 -0400
Modify: 2008-10-08 15:15:39.000000000 -0400
Change: 2008-10-08 15:15:39.000000000 -0400
```

如果我们再次 touch 这个文件，然后用 stat 命令检测它，我们会发现所有文件的时间已经更新了。

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
File: 'playground/timestamp'
Size: 0 Blocks: 0 IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1001/ me) Gid: ( 1001/ me)
Access: 2008-10-08 15:23:33.000000000 -0400
Modify: 2008-10-08 15:23:33.000000000 -0400
Change: 2008-10-08 15:23:33.000000000 -0400
```

下一步，让我们使用 find 命令来更新一些操练场中的文件：

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch '{}' ';'
```

这会更新操练场中所有名为 file-B 的文件。接下来我们会使用 find 命令来识别已更新的文件， 通过把所有文件与参考文件 timestamp 做比较：

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

搜索结果包含所有一百个文件 file-B 的实例。因为我们在更新了文件 timestamp 之后， touch 了操练场中名为 file-B 的所有文件，所以现在它们“新于”timestamp 文件，因此能被用 -newer 测试条件识别出来。

最后，让我们回到之前那个错误权限的例子中，把它应用于操练场里：

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \( -type d -not -perm 0700 \)
```

这个命令列出了操练场中所有一百个目录和二百六十个文件（还有 timestamp 和操练场本身，共 2702 个），因为没有一个符合我们“正确权限”的定义。通过对运算符和行为知识的了解，我们可以给这个命令 添加行为，对实战场中的文件和目录应用新的权限。

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600 '{}' ';' \)
   -or \( -type d -not -perm 0711 -exec chmod 0700 '{}' ';' \)
```

在日常的基础上，我们可能发现运行两个命令会比较容易一些，一个操作目录，另一个操作文件， 而不是这一个长长的复合命令，但是很高兴知道，我们能这样执行命令。这里最重要的一点是要 理解怎样把操作符和行为结合起来使用，来执行有用的任务。

选项

最后，我们有这些选项。这些选项被用来控制 find 命令的搜索范围。当构建 find 表达式的时候， 它们可能被其它的测试条件和行为包含：

表 18-7: find 命令选项

选项	描述
-depth	指导 find 程序先处理目录中的文件，再处理目录自身。当指定-delete 行为时，会自动应用这个选项。
-maxdepth levels	当执行测试条件和行为的时候，设置 find 程序陷入目录树的最大级别数
-mindepth levels	在应用测试条件和行为之前，设置 find 程序陷入目录数的最小级别数。
-mount	指导 find 程序不要搜索挂载到其它文件系统上的目录。
-noleaf	指导 find 程序不要基于搜索类似于 Unix 的文件系统做出的假设，来优化它的搜索。

拓展阅读

- 程序 locate，updatedb，find 和 xargs 都是 GNU 项目 findutils 软件包的一部分。 这个 GUN 项目提供了大量的在线文档，这些文档相当出色，如果你在高安全性的 环境中使用这些程序，你应该读读这些文档。

  http://www.gnu.org/software/findutils/

layout: book-zh

title: 归档和备份

计算机系统管理员的一个主要任务就是保护系统的数据安全。一种方法是通过时时备份系统文件，来保护 数据。即使你不是一名系统管理员，复制东西，在各个位置和设备之间移动大量的文件，也是很有帮助的。 在这一章中，我们将会看看几个经常用来管理文件集合的程序。它们就是文件压缩程序：

- gzip – 压缩或者展开文件
- bzip2 – 块排序文件压缩器

归档程序：

- tar – 磁带打包工具
- zip – 打包和压缩文件

还有文件同步程序：

- rsync – 同步远端文件和目录

压缩文件

纵观计算领域的发展历史，人们努力想把最多的数据存放到到最小的可用空间中，不管是内存，存储设备 还是网络带宽。今天我们把许多数据服务都看作是理所当然的事情，但是诸如便携式音乐播放器， 高清电视，或宽带网络之类的存在都应归功于高效的数据压缩技术。

数据压缩就是一个删除冗余数据的过程。让我们考虑一个假想的例子，比方说我们有一张100*100像素的 纯黑的图片文件。根据数据存储方案（假定每个像素占24位，或者3个字节），那么这张图像将会占用 30,000个字节的存储空间：

```
100 * 100 * 3 = 30,000
```

一张单色图像包含的数据全是多余的。我们要是聪明的话，可以用这种方法来编码这些数据， 我们只要简单地描述这个事实，我们有3万个黑色的像素数据块。所以，我们不存储包含3万个0（通常在图像文件中，黑色由0来表示）的数据块，取而代之，我们把这些数据压缩为数字30,000， 后跟一个0，来表示我们的数据。这种数据压缩方案被称为游程编码，是一种最基本的压缩技术。

压缩算法（数学技巧被用来执行压缩任务）分为两大类，无损压缩和有损压缩。无损压缩保留了 原始文件的所有数据。这意味着，当还原一个压缩文件的时候，还原的文件与原文件一模一样。 而另一方面，有损压缩，执行压缩操作时会删除数据，允许更大的压缩。当一个有损文件被还原的时候， 它与原文件不相匹配; 相反，它是一个近似值。有损压缩的例子有 JPEG（图像）文件和 MP3（音频）文件。 在我们的讨论中，我们将看看完全无损压缩，因为计算机中的大多数数据是不能容忍丢失任何数据的。

这个 gzip 程序被用来压缩一个或多个文件。当执行 gzip 命令时，则原始文件的压缩版会替代原始文件。 相对应的 gunzip 程序被用来把压缩文件复原为没有被压缩的版本。这里有个例子：

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me     me 15738 2008-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me     me 3230 2008-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me     me 15738 2008-10-14 07:15 foo.txt
```

在这个例子里，我们创建了一个名为 foo.txt 的文本文件，其内容包含一个目录的列表清单。 接下来，我们运行 gzip 命令，它会把原始文件替换为一个叫做 foo.txt.gz 的压缩文件。在 foo.*文件列表中，我们看到原始文件已经被压缩文件替代了，并将这个压缩文件大约是原始 文件的十五分之一。我们也能看到压缩文件与原始文件有着相同的权限和时间戳。

接下来，我们运行 gunzip 程序来解压缩文件。随后，我们能见到压缩文件已经被原始文件替代了， 同样地保留了相同的权限和时间戳。

gzip 命令有许多选项。这里列出了一些：

表19-1: gzip 选项

选项	说明
-c	把输出写入到标准输出，并且保留原始文件。也有可能用--stdout 和--to-stdout 选项来指定。
-d	解压缩。正如 gunzip 命令一样。也可以用--decompress 或者--uncompress 选项来指定.
-f	强制压缩，即使原始文件的压缩文件已经存在了，也要执行。也可以用--force 选项来指定。
-h	显示用法信息。也可用--help 选项来指定。
-l	列出每个被压缩文件的压缩数据。也可用--list 选项。
-r	若命令的一个或多个参数是目录，则递归地压缩目录中的文件。也可用--recursive 选项来指定。
-t	测试压缩文件的完整性。也可用--test 选项来指定。
-v	显示压缩过程中的信息。也可用--verbose 选项来指定。
-number	设置压缩指数。number 是一个在1（最快，最小压缩）到9（最慢，最大压缩）之间的整数。 数值1和9也可以各自用--fast 和--best 选项来表示。默认值是整数6。

返回到我们之前的例子中：

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz: OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

这里，我们用压缩文件来替代文件 foo.txt，压缩文件名为 foo.txt.gz。下一步，我们测试了压缩文件 的完整性，使用了-t 和-v 选项。

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

这个命令创建了一个目录列表的压缩文件。

这个 gunzip 程序，会解压缩 gzip 文件，假定那些文件名的扩展名是.gz，所以没有必要指定它， 只要指定的名字与现有的未压缩文件不冲突就可以：

```
[me@linuxbox ~]$ gunzip foo.txt
```

如果我们的目标只是为了浏览一下压缩文本文件的内容，我们可以这样做：

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

另外，对应于 gzip 还有一个程序，叫做 zcat，它等同于带有-c 选项的 gunzip 命令。 它可以被用来如 cat 命令作用于 gzip 压缩文件：

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

小贴士: 还有一个 zless 程序。它与上面的管道线有相同的功能。

这个 bzip2 程序，由 Julian Seward 开发，与 gzip 程序相似，但是使用了不同的压缩算法， 舍弃了压缩速度，而实现了更高的压缩级别。在大多数情况下，它的工作模式等同于 gzip。 由 bzip2 压缩的文件，用扩展名 .bz2 来表示：

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me     me      15738 2008-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me     me      2792 2008-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

正如我们所看到的，bzip2 程序使用起来和 gzip 程序一样。我们之前讨论的 gzip 程序的所有选项（除了-r），bzip2 程序同样也支持。注意，然而，压缩级别选项（-number）对于 bzip2 程序来说，有少许不同的含义。 伴随着 bzip2 程序，有 bunzip2 和 bzcat 程序来解压缩文件。bzip2 文件也带有 bzip2recover 程序，其会 试图恢复受损的.bz2 文件。

> 不要强迫性压缩
>
> 我偶然见到人们试图用高效的压缩算法，来压缩一个已经被压缩过的文件，通过这样做：
>
> *$ gzip picture.jpg*
>
> 不要这样。你可能只是在浪费时间和空间！如果你再次压缩已经压缩过的文件，实际上你 会得到一个更大的文件。这是因为所有的压缩技术都会涉及一些开销，文件中会被添加描述 此次压缩过程的信息。如果你试图压缩一个已经不包含多余信息的文件，那么再次压缩不会节省 空间，以抵消额外的花费。 {: .single}

归档文件

一个常见的，与文件压缩结合一块使用的文件管理任务是归档。归档就是收集许多文件，并把它们 捆绑成一个大文件的过程。归档经常作为系统备份的一部分来使用。当把旧数据从一个系统移到某 种类型的长期存储设备中时，也会用到归档程序。

在类似于 Unix 的软件世界中，这个 tar 程序是用来归档文件的经典工具。它的名字，是 tape archive 的简称，揭示了它的根源，它是一款制作磁带备份的工具。而它仍然被用来完成传统任务， 它也同样适用于其它的存储设备。我们经常看到扩展名为.tar 或者.tgz 的文件，它们各自表示“普通” 的 tar 包和被 gzip 程序压缩过的 tar 包。一个 tar 包可以由一组独立的文件，一个或者多个目录，或者 两者混合体组成。命令语法如下：

这里的 mode 是指以下操作模式（这里只展示了一部分，查看 tar 的手册来得到完整列表）之一：

表19-2: tar 模式

模式	说明
c	为文件和／或目录列表创建归档文件。
x	抽取归档文件。
r	追加具体的路径到归档文件的末尾。
t	列出归档文件的内容。

tar 命令使用了稍微有点奇怪的方式来表达它的选项，所以我们需要一些例子来展示它是 怎样工作的。首先，让我们重新创建之前我们用过的操练场:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A-Z}
```

下一步，让我们创建整个操练场的 tar 包：

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

这个命令创建了一个名为 playground.tar 的 tar 包，其包含整个 playground 目录层次结果。我们 可以看到模式 c 和选项 f，其被用来指定这个 tar 包的名字，模式和选项可以写在一起，而且不 需要开头的短横线。注意，然而，必须首先指定模式，然后才是其它的选项。

要想列出归档文件的内容，我们可以这样做：

```
[me@linuxbox ~]$ tar tf playground.tar
```

为了得到更详细的列表信息，我们可以添加选项 v：

```
[me@linuxbox ~]$ tar tvf playground.tar
```

现在，抽取 tar 包 playground 到一个新位置。我们先创建一个名为 foo 的新目录，更改目录， 然后抽取 tar 包中的文件：

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox ~]$ tar xf ../playground.tar
[me@linuxbox ~]$ ls
playground
```

如果我们检查~/foo/playground 目录中的内容，会看到这个归档文件已经被成功地安装了，就是创建了 一个精确的原始文件的副本。有一个警告，然而：除非你是超级用户，要不然从归档文件中抽取的文件 和目录的所有权由执行此复原操作的用户所拥有，而不属于原始所有者。

tar 命令另一个有趣的行为是它处理归档文件路径名的方式。默认情况下，路径名是相对的，而不是绝对 路径。当创建归档文件的时候，tar 命令会简单地删除路径名开头的斜杠。为了说明问题，我们将会 重新创建我们的归档文件，这次指定一个绝对路径：

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

记住，当按下回车键后，~/playground 会展开成/home/me/playground，所以我们将会得到一个 绝对路径名。接下来，和之前一样我们会抽取归档文件，观察发生什么事情：

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home     playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

这里我们看到当我们抽取第二个归档文件时，它重新创建了 home/me/playground 目录， 相对于我们当前的工作目录，~/foo，而不是相对于 root 目录，作为带有绝对路径名的案例。 这看起来似乎是一种奇怪的工作方式，但事实上这种方式很有用，因为这样就允许我们抽取文件 到任意位置，而不是强制地把抽取的文件放置到原始目录下。加上 verbose（v）选

项，重做 这个练习，将会展现更加详细的信息。

让我们考虑一个假设，tar 命令的实际应用。假定我们想要复制主目录及其内容到另一个系统中， 并且有一个大容量的 USB 硬盘，可以把它作为传输工具。在现代 Linux 系统中， 这个硬盘会被“自动地”挂载到/media 目录下。我们也假定硬盘中有一个名为 BigDisk 的逻辑卷。 为了制作 tar 包，我们可以这样做：

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

tar 包制作完成之后，我们卸载硬盘，然后把它连接到第二个计算机上。再一次，此硬盘被 挂载到/media/BigDisk 目录下。为了抽取归档文件，我们这样做：

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

值得注意的一点是，因为归档文件中的所有路径名都是相对的，所以首先我们必须更改目录到根目录下， 这样抽取的文件路径就相对于根目录了。

当抽取一个归档文件时，有可能限制从归档文件中抽取什么内容。例如，如果我们想要抽取单个文件， 可以这样实现：

```
tar xf archive.tar pathname
```

通过给命令添加末尾的路径名，tar 命令就只会恢复指定的文件。可以指定多个路径名。注意 路径名必须是完全的，精准的相对路径名，就如存储在归档文件中的一样。当指定路径名的时候， 通常不支持通配符；然而，GNU 版本的 tar 命令（在 Linux 发行版中最常出现）通过--wildcards 选项来 支持通配符。这个例子使用了之前 playground.tar 文件：

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/playground/dir-*/file-A'
```

这个命令将只会抽取匹配特定路径名的文件，路径名中包含了通配符 dir-*。

tar 命令经常结合 find 命令一起来制作归档文件。在这个例子里，我们将会使用 find 命令来 产生一个文件集合，然后这些文件被包含到归档文件中。

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf playground.tar '{}' '+'
```

这里我们使用 find 命令来匹配 playground 目录中所有名为 file-A 的文件，然后使用-exec 行为，来 唤醒带有追加模式（r）的 tar 命令，把匹配的文件添加到归档文件 playground.tar 里面。

使用 tar 和 find 命令，来创建逐渐增加的目录树或者整个系统的备份，是个不错的方法。通过 find 命令匹配新于某个时间戳的文件，我们就能够创建一个归档文件，其只包含新于上一个 tar 包的文件， 假定这个时间戳文件恰好在每个归档文件创建之后被更新了。

tar 命令也可以利用标准输出和输入。这里是一个完整的例子:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-from=-
   | gzip > playground.tgz
```

在这个例子里面，我们使用 find 程序产生了一个匹配文件列表，然后把它们管道到 tar 命令中。 如果指定了文件名“-”，则其被看作是标准输入或输出，正是所需（顺便说一下，使用“-”来表示 标准输入／输出的惯例，也被大量的其它程序使用）。这个--file-from 选项（也可以用-T 来指定） 导致 tar 命令从一个文件而不是命令行来读入它的路径名列表。最后，这个由 tar 命令产生的归档 文件被管道到 gzip 命令中，然后创建了压缩归档文件 playground.tgz。此.tgz 扩展名是命名 由 gzip 压缩的 tar

文件的常规扩展名。有时候也会使用.tar.gz 这个扩展名。

虽然我们使用 gzip 程序来制作我们的压缩归档文件，但是现在的 GUN 版本的 tar 命令， gzip 和 bzip2压缩两者都直接支持，各自使用 z 和 j 选项。以我们之前的例子为基础， 我们可以这样简化它：

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf playground.tgz -T -
```

如果我们本要创建一个由 bzip2压缩的归档文件，我们可以这样做：

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf playground.tbz -T -
```

通过简单地修改压缩选项，把 z 改为 j（并且把输出文件的扩展名改为.tbz，来指示一个 bzip2压缩文件）， 就使 bzip2命令压缩生效了。另一个 tar 命令与标准输入和输出的有趣使用，涉及到在系统之间经过 网络传输文件。假定我们有两台机器，每台都运行着类似于 Unix，且装备着 tar 和 ssh 工具的操作系统。 在这种情景下，我们可以把一个目录从远端系统（名为 remote-sys）传输到我们的本地系统中：

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar xf -
me@remote-sys’s password:
[me@linuxbox remote-stuff]$ ls
Documents
```

这里我们能够从远端系统 remote-sys 中复制目录 Documents 到本地系统名为 remote-stuff 目录中。 我们怎样做的呢？首先，通过使用 ssh 命令在远端系统中启动 tar 程序。你可记得 ssh 允许我们 在远程联网的计算机上执行程序，并且在本地系统中看到执行结果——远端系统中产生的输出结果 被发送到本地系统中查看。我们可以利用。在本地系统中，我们执行 tar 命令，

这个 zip 程序既是压缩工具，也是一个打包工具。这程序使用的文件格式，Windows 用户比较熟悉， 因为它读取和写入.zip 文件。然而，在 Linux 中 gzip 是主要的压缩程序，而 bzip2则位居第二。

在 zip 命令最基本的使用中，可以这样唤醒 zip 命令：

```
zip options zipfile file...
```

例如，制作一个 playground 的 zip 版本的文件包，这样做：

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

除非我们包含-r 选项，要不然只有 playground 目录（没有任何它的内容）被存储。虽然会自动添加 .zip 扩展名，但为了清晰起见，我们还是包含文件扩展名。

在创建 zip 版本的文件包时，zip 命令通常会显示一系列的信息：

```
adding: playground/dir-020/file-Z (stored 0%)
adding: playground/dir-020/file-Y (stored 0%)
adding: playground/dir-020/file-X (stored 0%)
adding: playground/dir-087/ (stored 0%)
adding: playground/dir-087/file-S (stored 0%)
```

这些信息显示了添加到文件包中每个文件的状态。zip 命令会使用两种存储方法之一，来添加 文件到文件包中：要不它会“store”没有压缩的文件，正如这里所示，或者它会“deflate”文件， 执行压缩操作。在存储方法之后显示的数值表明了压缩量。因为我们的 playground 目录 只是包含空文件，没有对它的内容执行压缩操作。

使用 unzip 程序，来直接抽取一个 zip 文件的内容。

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

对于 zip 命令（与 tar 命令相反）要注意一点，就是如果指定了一个已经存在的文件包，其被更新 而不是被替代。这意味着会保留此文件包，但是会添加新文件，同时替换匹配的文件。可以列出 文件或者有选择地从一个 zip 文件包中抽取文件，只要给 unzip 命令指定文件名：

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-87/file-Z
Archive: ../playground.zip
    Length      Date    Time    Name

        0    10-05-08  09:25   playground/dir-87/file-Z

        0                     1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ./playground.zip playground/dir-87/file-Z
Archive: ../playground.zip
replace playground/dir-87/file-Z? [y]es, [n]o, [A]ll, [N]one,
[r]ename: y
extracting: playground/dir-87/file-Z
```

使用-l 选项，导致 unzip 命令只是列出文件包中的内容而没有抽取文件。如果没有指定文件， unzip 程序将会列出文件包中的所有文件。添加这个-v 选项会增加列表的冗余信息。注意当抽取的 文件与已经存在的文件冲突时，会在替代此文件之前提醒用户。

像 tar 命令一样，zip 命令能够利用标准输入和输出，虽然它的实施不大有用。通过-@选项，有可能把一系列的 文件名管道到 zip 命令。

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

这里我们使用 find 命令产生一系列与“file-A”相匹配的文件列表，并且把此列表管道到 zip 命令， 然后创建包含所选文件的文件包 file-A.zip。

zip 命令也支持把它的输出写入到标准输出，但是它的使用是有限的，因为很少的程序能利用输出。 不幸地是，这个 unzip 程序，不接受标准输入。这就阻止了 zip 和 unzip 一块使用，像 tar 命令那样， 来复制网络上的文件。

然而，zip 命令可以接受标准输入，所以它可以被用来压缩其它程序的输出：

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
adding: - (deflated 80%)
```

在这个例子里，我们把 ls 命令的输出管道到 zip 命令。像 tar 命令，zip 命令把末尾的横杠解释为 “使用标准输入作为输入文件。”

这个 unzip 程序允许它的输出发送到标准输出，当指定了-p 选项之后：

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

我们讨论了一些 zip/unzip 可以完成的基本操作。它们两个都有许多选项，其增加了 命令的灵活性，虽然一些选项只针对于特定的平台。zip 和 unzip 命令的说明手册都相当不错， 并且包含了有用的实例。然而，这些程序的主要用途是为了和 Windows 系统交换文件， 而不是在 Linux 系统中执行压缩和打包操作，tar 和 gzip 程序在 Linux 系统中更受欢迎。

同步文件和目录

维护系统备份的常见策略是保持一个或多个目录与另一个本地系统（通常是某种可移动的存储设备） 或者远端系统中的目录（或多个目录）同步。我们可能，例如有一个正在开发的网站的本地备份， 需要时不时的与远端网络服务器中的文件备份保持同步。在类似于 Unix 系统的世界里，能完成此任务且 备受人们喜爱的工具是 rsync。这个程序能同步本地与远端的目录，通过使用 rsync 远端更新协议，此协议 允许 rsync 快速地检测两个目录的差异，执行最小量的复制来达到目录间的同步。比起其它种类的复制程序， 这就使 rsync 命令非常快速和高效。

rsync 被这样唤醒：

这里 source 和 destination 是下列选项之一：

- 一个本地文件或目录
- 一个远端文件或目录，以[user@]host:path 的形式存在
- 一个远端 rsync 服务器，由 rsync://[user@]host[:port]/path 指定

注意 source 和 destination 两者之一必须是本地文件。rsync 不支持远端到远端的复制

让我们试着对一些本地文件使用 rsync 命令。首先，清空我们的 foo 目录：

```
[me@linuxbox ~]$ rm -rf foo/*
```

下一步，我们将同步 playground 目录和它在 foo 目录中相对应的副本

```
[me@linuxbox ~]$ rsync -av playground foo
```

我们包括了-a 选项（递归和保护文件属性）和-v 选项（冗余输出）， 来在 foo 目录中制作一个 playground 目录的镜像。当这个命令执行的时候， 我们将会看到一系列的文件和目录被复制。在最后，我们将看到一条像这样的总结信息：

```
sent 135759 bytes received 57870 bytes 387258.00 bytes/sec
total size is 3230 speedup is 0.02
```

说明复制的数量。如果我们再次运行这个命令，我们将会看到不同的结果：

```
[me@linuxbox ~]$ rsync -av playgound foo
building file list ... done
sent 22635 bytes received 20 bytes
total size is 3230 speedup is 0.14
45310.00 bytes/sec
```

注意到没有文件列表。这是因为 rsync 程序检测到在目录~/playground 和 ~/foo/playground 之间 不存在差异，因此它不需要复制任何数据。如果我们在 playground 目录中修改一个文件，然后 再次运行 rsync 命令：

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes received 42 bytes 45454.00 bytes/sec
total size is 3230 speedup is 0.14
```

我们看到 rsync 命令检测到更改，并且只是复制了更新的文件。作为一个实际的例子， 让我们考虑一个假想的外部硬盘，之前我们在 tar 命令中用到过的。如果我们再次把此 硬盘连接到我们的系统中，它被挂载到/media/BigDisk 目录下，我们可以执行一个有 用的系统备份了，首先在外部硬盘上创建一个目录，名为/backup，然后使用 rsync 程序 从我们的系统中复制最重要的数据到此外部硬盘上：

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/backup
```

在这个例子里，我们把 /etc，/home，和 /usr/local 目录从我们的系统中复制到假想的存储设备中。 我们包含了--delete 这个选项，来删除可能在备份设备中已经存在但却不再存在于源设备中的文件，（这与我们第一次创建备份无关，但是会在随后的复制操作中有用途）。挂载外部驱动器，运行 rsync 命令，不断重复这个过程，是一个不错的（虽然不理想）方式来保存少量的系统备份文件。 当然，别名会对这个操作更有帮助些。我们将会创建一个别名，并把它添加到 .bashrc 文件中， 来提供这个特性：

```
alias backup='sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/backup'
```

现在我们所做的事情就是连接外部驱动器，然后运行 backup 命令来完成工作。

在网络间使用 rsync 命令

rsync 程序的真正好处之一，是它可以被用来在网络间复制文件。毕竟，rsync 中的“r”象征着“remote”。 远程复制可以通过两种方法完成。第一个方法要求另一个系统已经安装了 rsync 程序，还安装了 远程 shell 程序，比如 ssh。比方说我们本地网络中的一个系统有大量可用的硬盘空间，我们想要 用远程系统来代替一个外部驱动器，来执行文件备份操作。假定远程系统中有一个名为/backup 的目录， 其用来存放我们传送的文件，我们这样做：

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home /usr/local remote-sys:/backup
```

我们对命令做了两处修改，来方便网络间文件复制。首先，我们添加了--rsh=ssh 选项，其指示 rsync 使用 ssh 程序作为它的远程 shell。以这种方式，我们就能够使用一个 ssh 加密通道，把数据 安全地传送到远程主机中。其次，通过在目标路径名前加上远端主机的名字（在这种情况下， 远端主机名为 remote-sys），来指定远端主机。

rsync 可以被用来在网络间同步文件的第二种方式是通过使用 rsync 服务器。rsync 可以被配置为一个 守护进程， 监听即将到来的同步请求。这样做经常是为了允许一个远程系统的镜像。例如，Red Hat 软件中心为它的 Fedora 发行版，维护着一个巨大的正在开发中的软件包的仓库。对于软件测试人员， 在发行周期的测试阶段，镜像这些软件集合是非常有帮助的。因为仓库中的这些文件会频繁地 （通常每天不止一次）改动，定期同步本地镜像，这是可取的，而不是大量地拷贝软件仓库。 这些软件库之一被维护在 Georgia Tech；我们可以使用本地 rsync 程序和它们的 rsync 服务器来镜像它。

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av -delete rsync://rsync.gtlib.gatech.edu/fedora-linux-
 core/development/i386/os fedora-devel
```

在这个例子里，我们使用了远端 rsync 服务器的 URI，其由协议（rsync://），远端主机名 （rsync.gtlib.gatech.edu），和软件仓库的路径名组成。

拓展阅读

- 在这里讨论的所有命令的手册文档都相当清楚明白，并且包含了有用的例子。另外， GNU 版本的 tar 命令有一个不错的在线文档。可以在下面链接处找到：

  http://www.gnu.org/software/tar/manual/index.html

layout: book-zh

title: 正则表达式

接下来的几章中，我们将会看一下一些用来操作文本的工具。正如我们所见到的，在类似于 Unix 的 操作系统中，比如 Linux 中，文本数据起着举足轻重的作用。但是在我们能完全理解这些工具提供的 所有功能之前，我们不得不先看看，经常与这些工具的高级使用相关联的一门技术——正则表达式。

我们已经浏览了许多由命令行提供的功能和工具，我们遇到了一些真正神秘的 shell 功能和命令， 比如 shell 展开和引用，键盘快捷键，和命令历史，更不用说 vi 编辑器了。正则表达式延续了 这种"传统"，而且有可能（备受争议地）是其中最神秘的功能。这并不是说花费时间来学习它们 是不值得的，而是恰恰相反。虽然它们的全部价值可能不能立即显现，但是较强理解这些功能 使我们能够表演令人惊奇的技艺。什么是正则表达式？

简而言之，正则表达式是一种符号表示法，被用来识别文本模式。在某种程度上，它们与匹配 文件和路径名的 shell 通配符比较相似，但其规模更庞大。许多命令行工具和大多数的编程语言 都支持正则表达式，以此来帮助解决文本操作问题。然而，并不是所有的正则表达式都是一样的， 这就进一步混淆了事情；不同工具以及不同语言之间的正则表达式都略有差异。我们将会限定 POSIX 标准中描述的正则表达式（其包括了大多数的命令行工具），供我们讨论， 与许多编程语言（最著名的 Perl 语言）相反，它们使用了更多和更丰富的符号集。

我们将使用的主要程序是我们的老朋友，grep 程序，它会用到正则表达式。实际上，grep 这个名字 来自于短语 "global regular expression print"，所以我们能看出 grep 程序和正则表达式有关联。 本质上，grep 程序会在文本文件中查找一个指定的正则表达式，并把匹配行输出到标准输出。

到目前为止，我们已经使用 grep 程序查找了固定的字符串，就像这样:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

这个命令会列出，位于目录 /usr/bin 中，文件名中包含子字符串 zip 的所有文件。

这个 grep 程序以这样的方式来接受选项和参数：

```
grep [options] regex [file...]
```

这里的 regx 是指一个正则表达式。

这是一个常用的 grep 选项列表：

表20-1: grep 选项

选项	描述
-i	忽略大小写。不会区分大小写字符。也可用 --ignore-case 来指定。
-v	不匹配。通常，grep 程序会打印包含匹配项的文本行。这个选项导致 grep 程序 只会不包含匹配项的文本行。也可用 --invert-match 来指定。
-c	打印匹配的数量（或者是不匹配的数目，若指定了-v 选项），而不是文本行本身。 也可用 --count 选项来指定。
-l	打印包含匹配项的文件名，而不是文本行本身，也可用 --files-with-matches 选项来指定。
-L	相似于-l 选项，但是只是打印不包含匹配项的文件名。也可用 --files-without-match 来指定。
-n	在每个匹配行之前打印出其位于文件中的相应行号。也可用 --line-number 选项来指定。
-h	应用于多文件搜索，不输出文件名。也可用 --no-filename 选项来指定。

为了更好的探究 grep 程序，让我们创建一些文本文件来搜寻：

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt     dirlist-sbin.txt    dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

我们能够对我们的文件列表执行简单的搜索，像这样：

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

在这个例子里，grep 程序在所有列出的文件中搜索字符串 bzip，然后找到两个匹配项，其都在 文件 dirlist-bin.txt 中。如果我们只是对包含匹配项的文件列表，而不是对匹配项本身感兴趣 的话，我们可以指定-l 选项：

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

相反地，如果我们只想查看不包含匹配项的文件列表，我们可以这样操作：

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

它可能看起来不明显，但是我们的 grep 程序一直使用了正则表达式， 虽然是非常简单的例子。 这个正则表达式 bzip 意味着，匹配项所在行至少包含4个字符，并且按照字符 b, z, i, 和 p 的顺序 出现在匹配行的某处，字符之间没有其它的字符。字符串 bzip 中的所有字符都是原义字符，因为 它们匹配本身。除了原义字符之外，正则表达式也可能包含元字符，其被用来指定更复杂的匹配项。 正则表达式元字符由以下字符组成：

```
^ $ . [ ] { } - ? * + ( ) | \
```

然后其它所有字符都被认为是原义字符，虽然在个别情况下，反斜杠会被用来创建元序列， 也允许元字符被转义为原义字符，而不是被解释为元字符。

注意：正如我们所见到的，当 shell 执行展开的时候，许多正则表达式元字符，也是对 shell 有特殊 含义的字符。当我们在命令行中传递包含元字符的正则表达式的时候，把元字符用引号引起来至关重要， 这样可以阻止 shell 试图展开它们。

我们将要查看的第一个元字符是圆点字符，其被用来匹配任意字符。如果我们在正则表达式中包含它， 它将会匹配在此位置的任意一个字符。这里有个例子：

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

我们在文件中查找包含正则表达式 .zip 的文本行。对于搜索结果，有几点需要注意一下。 注意没有找到这个 zip 程序。这是因为在我们的正则表达式中包含的圆点字符把所要求的匹配项的长度 增加到四个字符，并且字符串 zip 只包含三个字符，所以这个 zip 程序不匹配。另外，如果我们的文件列表 中有一些文件的扩展名是 .zip，则它们也会成为匹配项，因为文件扩展名中的圆点符号也会被看作是 “任意字符”。

在正则表达式中，插入符号和美元符号被看作是锚（定位点）。这意味着正则表达式 只有在文本行的开头或末尾被找到时，才算发生一次匹配。

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

这里我们分别在文件列表中搜索行首，行尾以及行首和行尾同时包含字符串 zip（例如，zip 独占一行）的匹配行。 注意正则表达式 ‘^$’（行首和行尾之间没有字符）会匹配空行。

> 字谜助手
>
> 到目前为止，甚至凭借我们有限的正则表达式知识，我们已经能做些有意义的事情了。
>
> 我妻子喜欢玩字谜游戏，有时候她会因为一个特殊的问题，而向我求助。类似这样的问题，“一个 有五个字母的单词，它的第三个字母是 ‘j’，最后一个字母是 ‘r’，是哪个单词？”这类问题会 让我动脑筋想想。
>
> 你知道你的 Linux 系统中带有一本英文字典吗？千真万确。看一下 /usr/share/dict 目录，你就能找到一本， 或几本。存储在此目录下的字典文件，其内容仅仅是一个长长的单词列表，每行一个单词，按照字母顺序排列。在我的 系统中，这个文件仅包含98,000个单词。为了找到可能的上述字谜的答案，我们可以这样做：
>
> ```
> [me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
> Major
> major
> ```
>
> 使用这个正则表达式，我们能在我们的字典文件中查找到包含五个字母，且第三个字母 是“j”，最后一个字母是“r”的所有单词。 {: .single}

中括号表达式和字符类

除了能够在正则表达式中的给定位置匹配任意字符之外，通过使用中括号表达式， 我们也能够从一个指定的字符集合中匹配一个单个的字符。通过中括号表达式，我们能够指定 一个字符集合（包含在不加中括号的情况下会被解释为元字符的字符）来被匹配。在这个例子里，使用了一个两个字符的集合：

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

我们匹配包含字符串“bzip”或者“gzip”的任意行。

一个字符集合可能包含任意多个字符，并且元字符被放置到中括号里面后会失去了它们的特殊含义。 然而，在两种情况下，会在中括号表达式中使用元字符，并且有着不同的含义。第一个元字符 是插入字符，其被用来表示否定；第二个是连字符字符，其被用来表示一个字符区域。

如果在正则表示式中的第一个字符是一个插入字符，则剩余的字符被看作是不会在给定的字符位置出现的 字符集合。通过修改之前的例子，我们试验一下：

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

通过激活否定操作，我们得到一个文件列表，它们的文件名都包含字符串 zip，并且 zip 的前一个字符 是除了 b 和 g 之外的任意字符。注意文件 zip 没有被发现。一个否定的字符集仍然在给定位置要求一个字符， 但是这个字符必须不是否定字符集的成员。

这个插入字符如果是中括号表达式中的第一个字符的时候，才会唤醒否定功能；否则，它会失去 它的特殊含义，变成字符集中的一个普通字符。

传统的字符区域

如果我们想要构建一个正则表达式，它可以在我们的列表中找到每个以大写字母开头的文件，我们 可以这样做：

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

这只是一个在正则表达式中输入26个大写字母的问题。但是输入所有字母非常令人烦恼，所以有另外一种方式：

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

通过使用一个三字符区域，我们能够缩写26个字母。任意字符的区域都能按照这种方式表达，包括多个区域， 比如下面这个表达式就匹配了所有以字母和数字开头的文件名：

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

在字符区域中，我们看到这个连字符被特殊对待，所以我们怎样在一个正则表达式中包含一个连字符呢？方法就是使连字符成为表达式中的第一个字符。考虑一下这两个例子：

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

这会匹配包含一个大写字母的文件名。然而：

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

上面的表达式会匹配包含一个连字符，或一个大写字母“A”，或一个大写字母“Z”的文件名。

POSIX 字符集

这个传统的字符区域在处理快速地指定字符集合的问题方面，是一个易于理解的和有效的方式。 不幸地是，它们不总是工作。到目前为止，虽然我们在使用 grep 程序的时候没有遇到任何问题， 但是我们可能在使用其它程序的时候会遭遇困难。

回到第5章，我们看看通配符怎样被用来完成路径名展开操作。在那次讨论中，我们说过在 某种程度上，那个字符区域被使用的方式几乎与在正则表达式中的用法一样，但是有一个问题：

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

（依赖于不同的 Linux 发行版，我们将得到不同的文件列表，有可能是一个空列表。这个例子来自于 Ubuntu） 这个命令产生了期望的结果——只有以大写字母开头的文件名，但是：

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

通过这个命令我们得到整个不同的结果（只显示了一部分结果列表）。为什么会是那样？ 说来话长，但是这个版本比较简短：

追溯到 Unix 刚刚开发的时候，它只知道 ASCII 字符，并且这个特性反映了事实。在 ASCII 中，前32个字符 （数字0－31）都是控制码（如 tabs，backspaces，和回车）。随后的32个字符（32－63）包含可打印的字符， 包括大多数的标点符号和数字0到9。再随后的32个字符（64－95）包含大写字符和一些更多的标点符号。 最后的31个字符（96－127）包含小写字母和更多的标点符号。基于这种安排方式，系统使用这种排序规则 的 ASCII：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

这个不同于正常的字典顺序，其像这样：

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

随着 Unix 系统的知名度在美国之外的国家传播开来，就需要支持不在 U.S.英语范围内的字符。 于是就扩展了这个 ASCII 字符表，使用了整个8位，添加了字符（数字128－255），这样就 容纳了更多的语言。

为了支持这种能力，POSIX 标准介绍了一种叫做 locale 的概念，其可以被调整，来为某个特殊的区域， 选择所需的字符集。通过使用下面这个命令，我们能够查看到我们系统的语言设置：

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

通过这个设置，POSIX 相容的应用程序将会使用字典排列顺序而不是 ASCII 顺序。这就解释了上述命令的行为。 当 [A-Z] 字符区域按照字典顺序解释的时候，包含除了小写字母 a 之外的所有字母，因此得到这样的结果。

为了部分地解决这个问题，POSIX 标准包含了大量的字符集，其提供了有用的字符区域。 下表中描述了它们：

表20-2: POSIX 字符集

字符集	说明
[:alnum:]	字母数字字符。在 ASCII 中，等价于：[A-Za-z0-9]
[:word:]	与[:alnum:]相同, 但增加了下划线字符。
[:alpha:]	字母字符。在 ASCII 中，等价于：[A-Za-z]
[:blank:]	包含空格和 tab 字符。
[:cntrl:]	ASCII 的控制码。包含了0到31，和127的 ASCII 字符。
[:digit:]	数字0到9
[:graph:]	可视字符。在 ASCII 中，它包含33到126的字符。
[:lower:]	小写字母。
[:punct:]	标点符号字符。在 ASCII 中，等价于：
[:print:]	可打印的字符。在[:graph:]中的所有字符，再加上空格字符。
[:space:]	空白字符，包括空格，tab，回车，换行，vertical tab, 和 form feed.在 ASCII 中， 等价于：[\t\r\n\v\f]
[:upper:]	大写字母。
[:xdigit:]	用来表示十六进制数字的字符。在 ASCII 中，等价于：[0-9A-Fa-f]

甚至通过字符集，仍然没有便捷的方法来表达部分区域，比如[A-M]。

通过使用字符集，我们重做上述的例题，看到一个改进的结果：

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

记住，然而，这不是一个正则表达式的例子，而是 shell 正在执行路径名展开操作。我们在这里展示这个例子， 是因为 POSIX 规范的字符集适用于二者。

> 恢复到传统的排列顺序
>
> 通过改变环境变量 LANG 的值，你可以选择让你的系统使用传统的（ASCII）排列规则。如上所示，这个 LANG 变量包含了语种和字符集。这个值最初由你安装 Linux 系统时所选择的安装语言决定。
>
> 使用 locale 命令，来查看 locale 的设置。
>
> 把这个 LANG 变量设置为 POSIX，来更改 locale，使其使用传统的 Unix 行为。

> 注意这个改动使系统为它的字符集使用 U.S. 英语（更准确地说，ASCII），所以要确认一下这 是否是你真正想要的效果。通过把这条语句添加到你的 .bashrc 文件中，你可以使这个更改永久有效。

POSIX 基本的 Vs.扩展的正则表达式

就在我们认为这已经非常令人困惑了，我们却发现 POSIX 把正则表达式的实现分成了两类： 基本正则表达式（BRE）和扩展的正则表达式（ERE）。既服从 POSIX 规范又实现了 BRE 的任意应用程序，都支持我们目前研究的所有正则表达式特性。我们的 grep 程序就是其中一个。

BRE 和 ERE 之间有什么区别呢？这是关于元字符的问题。BRE 可以辨别以下元字符：

```
^ $ . [ ] *
```

其它的所有字符被认为是文本字符。ERE 添加了以下元字符（以及与其相关的功能）:

```
( ) { } ? + |
```

然而（这也是有趣的地方），在 BRE 中，字符 (，)，{，和 } 用反斜杠转义后，被看作是元字符, 相反在 ERE 中，在任意元字符之前加上反斜杠会导致其被看作是一个文本字符。在随后的讨论中将会涵盖 很多奇异的特性。

因为我们将要讨论的下一个特性是 ERE 的一部分，我们将要使用一个不同的 grep 程序。照惯例， 一直由 egrep 程序来执行这项操作，但是 GUN 版本的 grep 程序也支持扩展的正则表达式，当使用了-E 选项之后。

> 在 20 世纪 80 年代，Unix 成为一款非常流行的商业操作系统，但是到了1988年，Unix 世界 一片混乱。许多计算机制造商从 Unix 的创建者 AT&T 那里得到了许可的 Unix 源码，并且 供应各种版本的操作系统。然而，在他们努力创造产品差异化的同时，每个制造商都增加了 专用的更改和扩展。这就开始限制了软件的兼容性。
>
> 专有软件供应商一如既往，每个供应商都试图玩赢游戏“锁定”他们的客户。这个 Unix 历史上 的黑暗时代，就是今天众所周知的 “the Balkanization”。
>
> 然后进入 IEEE（ 电气与电子工程师协会 ）时代。在上世纪 80 年代中叶，IEEE 开始制定一套标准， 其将会定义 Unix 系统（ 以及类似于 Unix 的系统 ）如何执行。这些标准，正式成为 IEEE 1003， 定义了应用程序编程接口（ APIs ）， shell 和一些实用程序，其将会在标准的类似于 Unix 操作系统中找到。“POSIX” 这个名字，象征着可移植的操作系统接口（为了额外的，添加末尾的 “X” ）， 是由 Richard Stallman 建议的（ 是的，的确是 Richard Stallman ）， 后来被 IEEE 采纳。

我们将要讨论的扩展表达式的第一个特性叫做 alternation（交替），其是一款允许从一系列表达式 之间选择匹配项的实用程序。就像中括号表达式允许从一系列指定的字符之间匹配单个字符那样， alternation 允许从一系列字符串或者是其它的正则表达式中选择匹配项。为了说明问题， 我们将会结合 echo 程序来使用 grep 命令。首先，让我们试一个普通的字符串匹配：

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

一个相当直截了当的例子，我们把 echo 的输出管道给 grep，然后看到输出结果。当出现 一个匹配项时，我们看到它会打印出来；当没有匹配项时，我们看到没有输出结果。

现在我们将添加 alternation，以竖杠线元字符为标记：

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

这里我们看到正则表达式 'AAA|BBB'，这意味着“匹配字符串 AAA 或者是字符串 BBB”。注意因为这是 一个扩展的特性，我们给 grep 命令（虽然我们能以 egrep 程序来代替）添加了 -E 选项，并且我们 把这个正则表达式用单引号引起来，为的是阻止 shell 把竖杠线元字符解释为一个 pipe 操作符。 Alternation 并不局限于两种选择：

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

为了把 alternation 和其它正则表达式元素结合起来，我们可以使用()来分离 alternation。

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

这个表达式将会在我们的列表中匹配以 bz，或 gz，或 zip 开头的文件名。如果我们删除了圆括号， 这个表达式的意思：

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

会变成匹配任意以 bz 开头，或包含 gz，或包含 zip 的文件名。

限定符

扩展的正则表达式支持几种方法，来指定一个元素被匹配的次数。

? - 匹配一个元素零次或一次

这个限定符意味着，实际上，“使前面的元素可有可无。”比方说我们想要查看一个电话号码的真实性， 如果它匹配下面两种格式的任意一种，我们就认为这个电话号码是真实的：

```
(nnn) nnn-nnnn

nnn nnn-nnnn
```

这里的“n”是一个数字。我们可以构建一个像这样的正则表达式：

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

在这个表达式中，我们在圆括号之后加上一个问号，来表示它们将被匹配零次或一次。再一次，因为 通常圆括号都是元字符（在 ERE 中），所以我们在圆括号之前加上了反斜杠，使它们成为文本字符。

让我们试一下：

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]
\)? [0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

这里我们看到这个表达式匹配这个电话号码的两种形式，但是不匹配包含非数字字符的号码。

像?元字符一样，这个 * 被用来表示一个可选的字符；然而，又与?不同，匹配的字符可以出现 任意多次，不仅是一次。比方说我们想要知道是否一个字符串是一句话；也就是说，字符串开始于 一个大写字母，然后包含任意多个大写和小写的字母和空格，最后以句号收尾。为了匹配这个（非常粗略的） 语句的定义，我们能够使用一个像这样的正则表达式：

```
[[:upper:]][[:upper:][:lower:] ]*.
```

这个表达式由三个元素组成：一个包含 [:upper:] 字符集的中括号表达式，一个包含 [:upper:] 和 [:lower:] 两个字符集以及一个空格的中括号表达式，和一个被反斜杠字符转义过的圆点。第二个元素末尾带有一个 * 元字符，所以在开头的大写字母之后，可能会跟随着任意数目的大写和小写字母和空格，并且匹配：

```
[me@linuxbox ~]$ echo "This works." | grep -E '[[:upper:]][[:upper:][[:lower:]]*.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '[[:upper:]][[:upper:][[:lower:]]*.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '[[:upper:]][[:upper: ][[:lower:]]*.'
[me@linuxbox ~]$
```

这个表达式匹配前两个测试语句，但不匹配第三个，因为第三个句子缺少开头的大写字母和末尾的句号。

+ - 匹配一个元素一次或多次

这个+元字符的作用与 * 非常相似，除了它要求前面的元素至少出现一次匹配。这个正则表达式只匹配 那些由一个或多个字母字符组构成的文本行，字母字符之间由单个空格分开：

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc  d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

我们看到这个正则表达式不匹配 “a b 9” 这一行，因为它包含了一个非字母的字符；它也不匹配 “abc d” ，因为在字符 c 和 d 之间不止一个空格。

{ } - 匹配一个元素特定的次数

{和}元字符都被用来表达要求匹配的最小和最大数目。它们可以通过四种方法来指定：

表20-3: 指定匹配的数目

限定符	意思
{n}	匹配前面的元素，如果它确切地出现了 n 次。
{n,m}	匹配前面的元素，如果它至少出现了 n 次，但是不多于 m 次。
{n,}	匹配前面的元素，如果它出现了 n 次或多于 n 次。

{,m}	匹配前面的元素，如果它出现的次数不多于 m 次。

回到之前处理电话号码的例子，我们能够使用这种指定重复次数的方法来简化我们最初的正则表达式：

```
^\(?[0-9][0-9][0-9]\)?  [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$

^\(?[0-9]{3}\)?  [0-9]{3}-[0-9]{4}$
```

让我们试一下：

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

我们可以看到，我们修订的表达式能成功地验证带有和不带有圆括号的数字，而拒绝那些格式 不正确的数字。

让正则表达式工作起来

让我们看看一些我们已经知道的命令，然后看一下它们怎样使用正则表达式。

通过 **grep** 命令来验证一个电话簿

在我们先前的例子中，我们查看过单个电话号码，并且检查了它们的格式。一个更现实的 情形是检查一个数字列表，所以我们先创建一个列表。我们将背诵一个神奇的咒语到命令行中。 它会很神奇，因为我们还没有涵盖所涉及的大部分命令，但是不要担心。我们将在后面的章节里面 讨论那些命令。这里是这个咒语：

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDO
M:0:3}-${RANDOM:0:4}" >> phonelist.txt; done
```

这个命令会创建一个包含10个电话号码的名为 phonelist.txt 的文件。每次重复这个命令的时候， 另外10个号码会被添加到这个列表中。我们也能够更改命令开头附近的数值10，来生成或多或少的 电话号码。如果我们查看这个文件的内容，然而我们会发现一个问题：

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

一些号码是残缺不全的，但是它们很适合我们的需求，因为我们将使用 grep 命令来验证它们。

一个有用的验证方法是扫描这个文件，查找无效的号码，并把搜索结果显示到屏幕上：

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$'
phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

这里我们使用 -v 选项来产生相反的匹配，因此我们将只输出不匹配指定表达式的文本行。这个 表达式自身的两端都包含定位点（锚）元字符，是为了确保这个号码的两端没有多余的字符。 这个表达式也要求圆括号出现在一个有效的号码中，不同于我们先前电话号码的实例。

用 **find** 查找丑陋的文件名

这个 find 命令支持一个基于正则表达式的测试。当在使用正则表达式方面比较 find 和 grep 命令的时候， 还有一个重要问题要牢记在心。当某一行包含的字符串匹配上了一个表达式的时候，grep 命令会打印出这一行， 然而 find 命令要求路径名精确地匹配这个正则表达式。在下面的例子里面，我们将使用带有一个正则 表达式的 find 命令，来查找每个路径名，其包含的任意字符都不是以下字符集中的一员。

```
[-\_./0-9a-zA-Z]
```

这样一种扫描会发现包含空格和其它潜在不规范字符的路径名：

```
[me@linuxbox ~]$ find . -regex '.*[^-\_./0-9a-zA-Z].*'
```

由于要精确地匹配整个路径名，所以我们在表达式的两端使用了 .*，来匹配零个或多个字符。 在表达式中间，我们使用了否定的中括号表达式，其包含了我们一系列可接受的路径名字符。

用 **locate** 查找文件

这个 locate 程序支持基本的（--regexp 选项）和扩展的（--regex 选项）正则表达式。通过 locate 命令，我们能够执行许多与先前操作 dirlist 文件时相同的操作：

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

通过使用 alternation，我们搜索包含 bin/bz，bin/gz，或/bin/zip 字符串的路径名。

在 **less** 和 **vim** 中查找文本

less 和 vim 两者享有相同的文本查找方法。按下/按键，然后输入正则表达式，来执行搜索任务。 如果我们使用 less 程序来浏览我们的 phonelist.txt 文件：

```
[me@linuxbox ~]$ less phonelist.txt
```

然后查找我们有效的表达式：

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

less 将会高亮匹配到的字符串，这样就很容易看到无效的电话号码：

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
(END)
```

另一方面，vim 支持基本的正则表达式，所以我们用于搜索的表达式看起来像这样：

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

我们看到表达式几乎一样；然而，在扩展表达式中，许多被认为是元字符的字符在基本的表达式 中被看作是文本字符。只有用反斜杠把它们转义之后，它们才被看作是元字符。

依赖于系统中 vim 的特殊配置，匹配项将会被高亮。如若不是，试试这个命令模式：

```
:hlsearch
```

来激活搜索高亮功能。

注意：依赖于你的发行版，vim 有可能支持或不支持文本搜索高亮功能。尤其是 Ubuntu 自带了 一款非常简化的 vim 版本。在这样的系统中，你可能要使用你的软件包管理器来安装一个功能 更完备的 vim 版本。

总结归纳

在这章中，我们已经看到几个使用正则表达式例子。如果我们使用正则表达式来搜索那些使用正则表达式的应用程序， 我们可以找到更多的使用实例。通过查找手册页，我们就能找到：

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

这个 zgrep 程序是 grep 的前端，允许 grep 来读取压缩文件。在我们的例子中，我们在手册文件所在的 目录中，搜索压缩文件中的内容。这个命令的结果是一个包含字符串 regex 或者 regular expression 的文件列表。正如我们所看到的，正则表达式会出现在大量程序中。

基本正则表达式中有一个特性，我们没有涵盖。叫做反引用，这个特性在下一章中会被讨论到。

拓展阅读

有许多在线学习正则表达式的资源，包括各种各样的教材和速记表。

另外，关于下面的背景话题，Wikipedia 有不错的文章。

- POSIX: http://en.wikipedia.org/wiki/Posix
- ASCII: http://en.wikipedia.org/wiki/Ascii

文本处理

所有类似于 Unix 的操作系统都非常依赖于被用于几种数据类型存储的文本文件。所以这很有道理， 有许多用于处理文本的工具。在这一章中，我们将看一些被用来“切割”文本的程序。在下一章中， 我们将查看更多的文本处理程序，但主要集中于文本格式化输出程序和其它一些人们需要的工具。

这一章会重新拜访一些老朋友，并且会给我们介绍一些新朋友：

- cat – 连接文件并且打印到标准输出
- sort – 给文本行排序
- uniq – 报告或者省略重复行
- cut – 从每行中删除文本区域
- paste – 合并文件文本行
- join – 基于某个共享字段来联合两个文件的文本行
- comm – 逐行比较两个有序的文件
- diff – 逐行比较文件
- patch – 给原始文件打补丁
- tr – 翻译或删除字符
- sed – 用于筛选和转换文本的流编辑器
- aspell – 交互式拼写检查器

文本应用程序

到目前为止，我们已经知道了一对文本编辑器（nano 和 vim），看过一堆配置文件，并且目睹了 许多命令的输出都是文本格式。但是文本还被用来做什么？ 它可以做很多事情。

许多人使用纯文本格式来编写文档。虽然很容易看到一个小的文本文件对于保存简单的笔记会 很有帮助，但是也有可能用文本格式来编写大的文档。一个流行的方法是先用文本格式来编写一个 大的文档，然后使用一种标记语言来描述已完成文档的格式。许多科学论文就是用这种方法编写的， 因为基于 Unix 的文本处理系统位于支持技术学科作家所需要的高级排版布局的一流系统之列。

世界上最流行的电子文档类型可能就是网页了。网页是文本文档，它们使用 HTML（超文本标记语言）或者是 XML （可扩展的标记语言）作为标记语言来描述文档的可视格式。

从本质上来说，email 是一个基于文本的媒介。为了传输，甚至非文本的附件也被转换成文本表示形式。 我们能看到这些，通过下载一个 email 信息，然后用 less 来浏览它。我们将会看到这条信息开始于一个标题， 其描述了信息的来源以及在传输过程中它接受到的处理，然后是信息的正文内容。

在类似于 Unix 的系统中，输出会以纯文本格式发送到打印机，或者如果页面包含图形，其会被转换成 一种文本格式的页面描述语言，以 PostScript 著称，然后再被发送给一款能产生图形点阵的程序， 最后被打印出来。

在类似于 Unix 系统中会发现许多命令行程序被用来支持系统管理和软件开发，并且文本处理程序也不例外。 许多文本处理程序被设计用来解决软件开发问题。文本处理对于软件开发者来言至关重要是因为所有的软件 都起始于文本格式。源代码，程序员实际编写的一部分程序，总是文本格式。

回到第7章（重定向），我们已经知道一些命令除了接受命令行参数之外，还能够接受标准输入。 那时候我们只是简单地介绍了它们，但是现在我们将仔细地看一下它们是怎样被用来执行文本处理的。

这个 cat 程序具有许多有趣的选项。其中许多选项用来帮助更好的可视化文本内容。一个例子是-A 选项， 其用来在文本中显示非打印字符。有些时候我们想知道是否控制字符嵌入到了我们的可见文本中。 最常用的控制字符是 tab 字符（而不是空格）和回车字符，在 MS-DOS 风格的文本文件中回车符经常作为 结束符出现。另一种常见情况是文件中包含末尾带有空格的文本行。

让我们创建一个测试文件，用 cat 程序作为一个简单的文字处理器。为此，我们将键入 cat 命令（随后指定了 用于重定向输出的文件），然后输入我们的文本，最后按下 Enter 键来结束这一行，然后按下组合键 Ctrl-d， 来指示 cat 程序，我们已经到达文件末尾了。在这个例子中，我们文本行的开头和末尾分别键入了一个 tab 字符以及一些空格。

```
[me@linuxbox ~]$ cat > foo.txt
	The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

下一步，我们将使用带有-A 选项的 cat 命令来显示这个文本：

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog.       $
[me@linuxbox ~]$
```

在输出结果中我们看到，这个 tab 字符在我们的文本中由^I 字符来表示。这是一种常见的表示方法，意思是 “Control-I”，结果证明，它和 tab 字符是一样的。我们也看到一个$字符出现在文本行真正的结尾处， 表明我们的文本包含末尾的空格。

> *MS-DOS 文本 Vs. Unix 文本*
>
> 可能你想用 cat 程序在文本中查看非打印字符的一个原因是发现隐藏的回车符。那么 隐藏的回车符来自于哪里呢？它们来自于 DOS 和 Windows！Unix 和 DOS 在文本文件中定义每行 结束的方式不相同。Unix 通过一个换行符（ASCII 10）来结束一行，然而 MS-DOS 和它的 衍生品使用回车（ASCII 13）和换行字符序列来终止每个文本行。
>
> 有几种方法能够把文件从 DOS 格式转变为 Unix 格式。在许多 Linux 系统中，有两个 程序叫做 dos2unix 和 unix2dos，它们能在两种格式之间转变文本文件。然而，如果你 的系统中没有安装 dos2unix 程序，也不要担心。文件从 DOS 格式转变为 Unix 格式的过程非常 简单；它只简单地涉及到删除违规的回车符。通过随后本章中讨论的一些程序，这个工作很容易完成。

cat 程序也包含用来修改文本的选项。最著名的两个选项是-n，其给文本行添加行号和-s， 禁止输出多个空白行。我们这样来说明：

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox


jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
1   The quick brown fox
2
3   jumped over the lazy dog.
[me@linuxbox ~]$
```

在这个例子里，我们创建了一个测试文件 foo.txt 的新版本，其包含两行文本，由两个空白行分开。 经由带有 -ns 选项的 cat 程序处理之后，多余的空白行被删除，并且对保留的文本行进行编号。 然而这并不是多个进程在操作这个文本，只有一个进程。

这个 sort 程序对标准输入的内容，或命令行中指定的一个或多个文件进行排序，然后把排序 结果发送到标准输出。使用与 cat 命令相同的技巧，我们能够演示如何用 sort 程序来处理标准输入：

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

输入命令之后，我们键入字母“c”，“b”，和“a”，然后再按下 Ctrl-d 组合键来表示文件的结尾。 随后我们查看生成的文件，看到文本行有序地显示。

因为 sort 程序能接受命令行中的多个文件作为参数，所以有可能把多个文件合并成一个有序的文件。例如， 如果我们有三个文本文件，想要把它们合并为一个有序的文件，我们可以这样做：

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

sort 程序有几个有趣的选项。这里只是一部分列表：

表21-1: 常见的 sort 程序选项

选项	长选项	描述
-b	--ignore-leading-blanks	默认情况下，对整行进行排序，从每行的第一个字符开始。这个选项导致 sort 程序忽略 每行开头的空格，从第一个非空白字符开始排序。
-f	--ignore-case	让排序不区分大小写。
-n	--numeric-sort	基于字符串的长度来排序。使用此选项允许根据数字值执行排序，而不是字母值。
-r	--reverse	按相反顺序排序。结果按照降序排列，而不是升序。
-k	--key=field1[,field2]	对从 field1到 field2之间的字符排序，而不是整个文本行。看下面的讨论。
-m	--merge	把每个参数看作是一个预先排好序的文件。把多个文件合并成一个排好序的文件，而没有执行额外的排序。
-o	--output=file	把排好序的输出结果发送到文件，而不是标准输出。
-t	--field-separator=char	定义域分隔字符。默认情况下，域由空格或制表符分隔。

虽然以上大多数选项的含义是不言自喻的，但是有些也不是。首先，让我们看一下 -n 选项，被用做数值排序。 通过这个选项，有可能基于数值进行排序。我们通过对 du 命令的输出结果排序来说明这个选项，du 命令可以 确定最大的磁盘空间用户。通常，这个 du 命令列出的输出结果按照路径名来排序：

```
[me@linuxbox ~]$ du -s /usr/share/\* | head
252     /usr/share/aclocal
96      /usr/share/acpi-support
8       /usr/share/adduser
196     /usr/share/alacarte
344     /usr/share/alsa
8       /usr/share/alsa-base
12488   /usr/share/anthy
8       /usr/share/apmd
21440   /usr/share/app-install
48      /usr/share/application-registry
```

在这个例子里面，我们把结果管道到 head 命令，把输出结果限制为前 10 行。我们能够产生一个按数值排序的 列表，来显示 10 个最大的空间消费者：

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940         /usr/share/locale-langpack
242660         /usr/share/doc
197560         /usr/share/fonts
179144         /usr/share/gnome
146764         /usr/share/myspell
144304         /usr/share/gimp
135880         /usr/share/dict
76508          /usr/share/icons
68072          /usr/share/apps
62844          /usr/share/foomatic
```

通过使用此 -nr 选项，我们产生了一个反向的数值排序，最大数值排列在第一位。这种排序起作用是 因为数值出现在每行的开头。但是如果我们想要基于文件行中的某个数值排序，又会怎样呢？ 例如，命令 ls -l 的输出结果：

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root   root     34824  2008-04-04  02:42 [
-rwxr-xr-x 1 root   root    101556  2007-11-27  06:08 a2p
...
```

此刻，忽略 ls 程序能按照文件大小对输出结果进行排序，我们也能够使用 sort 程序来完成此任务：

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nr -k 5 | head
-rwxr-xr-x 1 root   root   8234216  2008-04-0717:42 inkscape
-rwxr-xr-x 1 root   root   8222692  2008-04-07 17:42 inkview
...
```

sort 程序的许多用法都涉及到处理表格数据，例如上面 ls 命令的输出结果。如果我们 把数据库这个术语应用到上面的表格中，我们会说每行是一条记录，并且每条记录由多个字段组成， 例如文件属性，链接数，文件名，文件大小等等。sort 程序能够处理独立的字段。在数据库术语中， 我们能够指定一个或者多个关键字段，来作为排序的关键值。在上面的例子中，我们指定 n 和 r 选项来执行相反的数值排序，并且指定 -k 5，让 sort 程序使用第五字段作为排序的关键值。

这个 k 选项非常有趣，而且还有很多特点，但是首先我们需要讲讲 sort 程序怎样来定义字段。 让我们考虑一个非常简单的文本文件，只有一行包含作者名字的文本。

```
William      Shotts
```

默认情况下，sort 程序把此行看作有两个字段。第一个字段包含字符：

和第二个字段包含字符：

意味着空白字符（空格和制表符）被当作是字段间的界定符，当执行排序时，界定符会被 包含在字段当中。再看一下 ls 命令的输出，我们看到每行包含八个字段，并且第五个字段是文件大小：

```
-rwxr-xr-x 1 root root 8234216 2008-04-07 17:42 inkscape
```

让我们考虑用下面的文件，其包含从 2006 年到 2008 年三款流行的 Linux 发行版的发行历史，来做一系列实验。 文件中的每一行都有三个字段：发行版的名称，版本号，和 MM/DD/YYYY 格式的发行日期：

```
SUSE        10.2   12/07/2006
Fedora         10     11/25/2008
SUSE           11.04  06/19/2008
Ubuntu          8.04   04/24/2008
Fedora          8      11/08/2007
SUSE           10.3   10/04/2007
...
```

使用一个文本编辑器（可能是 vim），我们将输入这些数据，并把产生的文件命名为 distros.txt。

下一步，我们将试着对这个文件进行排序，并观察输出结果：

```
[me@linuxbox ~]$ sort distros.txt
Fedora          10     11/25/2008
Fedora          5     03/20/2006
Fedora          6     10/24/2006
Fedora          7     05/31/2007
Fedora          8     11/08/2007
...
```

恩，大部分正确。问题出现在 Fedora 的版本号上。因为在字符集中 "1" 出现在 "5" 之前，版本号 "10" 在 最顶端，然而版本号 "9" 却掉到底端。

为了解决这个问题，我们必须依赖多个键值来排序。我们想要对第一个字段执行字母排序，然后对 第三个字段执行数值排序。sort 程序允许多个 -k 选项的实例，所以可以指定多个排序关键值。事实上， 一个关键值可能包括一个字段区域。如果没有指定区域（如同之前的例子），sort 程序会使用一个键值， 其始于指定的字段，一直扩展到行尾。下面是多键值排序的语法：

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora         5     03/20/2006
Fedora         6     10/24/2006
Fedora         7     05/31/2007
...
```

虽然为了清晰，我们使用了选项的长格式，但是 -k 1,1 -k 2n 格式是等价的。在第一个 key 选项的实例中， 我们指定了一个字段区域。因为我们只想对第一个字段排序，我们指定了 1,1， 意味着"始于并且结束于第一个字段。"在第二个实例中，我们指定了 2n，意味着第二个字段是排序的键值， 并且按照数值排序。一个选项字母可能被包含在一个键值说明符的末尾，其用来指定排序的种类。这些 选项字母和 sort 程序的全局选项一样：b（忽略开头的空格），n（数值排序），r（逆向排序），等等。

我们列表中第三个字段包含的日期格式不利于排序。在计算机中，日期通常设置为 YYYY-MM-DD 格式， 这样使按时间顺序排序变得容易，但是我们的日期为美国格式 MM/DD/YYYY。那么我们怎样能按照 时间顺序来排列这个列表呢？

幸运地是，sort 程序提供了一种方式。这个 key 选项允许在字段中指定偏移量，所以我们能在字段中 定义键值。

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora         10    11/25/2008
Ubuntu         8.10  10/30/2008
SUSE           11.0  06/19/2008
...
```

通过指定 -k 3.7，我们指示 sort 程序使用一个排序键值，其始于第三个字段中的第七个字符，对应于 年的开头。同样地，我们指定 -k 3.1和 -k 3.4来分离日期中的月和日。 我们也添加了 n 和 r 选项来实现一个逆向的数值排序。这个 b 选项用来删除日期字段中开头的空格（ 行与行之间的空格数迥异，因此会影响 sort 程序的输出结果）。

一些文件不会使用 tabs 和空格做为字段界定符；例如，这个 /etc/passwd 文件：

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

这个文件的字段之间通过冒号分隔开，所以我们怎样使用一个 key 字段来排序这个文件？sort 程序提供 了一个 -t 选项来定义分隔符。按照第七个字段（帐户的默认 shell）来排序此 passwd 文件，我们可以这样做：

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

通过指定冒号字符做为字段分隔符，我们能按照第七个字段来排序。

与 sort 程序相比，这个 uniq 程序是个轻量级程序。uniq 执行一个看似琐碎的认为。当给定一个 排好序的文件（包括标准输出），uniq 会删除任意重复行，并且把结果发送到标准输出。 它常常和 sort 程序一块使用，来清理重复的输出。

---

uniq 程序是一个传统的 Unix 工具，经常与 sort 程序一块使用， 但是这个 GNU 版本的 sort 程序支持一个 -u 选项，其可以从排好序的输出结果中删除重复行。

让我们创建一个文本文件，来实验一下：

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

记住输入 Ctrl-d 来终止标准输入。现在，如果我们对文本文件执行 uniq 命令：

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
b
c
```

输出结果与原始文件没有差异；重复行没有被删除。实际上，uniq 程序能完成任务，其输入必须是排好序的数据，

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

这是因为 uniq 只会删除相邻的重复行。uniq 程序有几个选项。这里是一些常用选项：

表21-2: 常用的 uniq 选项

选项	说明
-c	输出所有的重复行，并且每行开头显示重复的次数。
-d	只输出重复行，而不是特有的文本行。
-f n	忽略每行开头的 n 个字段，字段之间由空格分隔，正如 sort 程序中的空格分隔符；然而， 不同于 sort 程序，uniq 没有选项来设置备用的字段分隔符。

-i	在比较文本行的时候忽略大小写。
-s n	跳过（忽略）每行开头的 n 个字符。
-u	只是输出独有的文本行。这是默认的。

这里我们看到 uniq 被用来报告文本文件中重复行的次数，使用这个-c 选项：

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
       2 a
       2 b
       2 c
```

下面我们将要讨论的三个程序用来从文件中获得文本列，并且以有用的方式重组它们。

这个 cut 程序被用来从文本行中抽取文本，并把其输出到标准输出。它能够接受多个文件参数或者 标准输入。

从文本行中指定要抽取的文本有些麻烦，使用以下选项：

表21-3: cut 程序选择项

选项	说明
-c char_list	从文本行中抽取由 char_list 定义的文本。这个列表可能由一个或多个逗号 分隔开的数值区间组成。
-f field_list	从文本行中抽取一个或多个由 field_list 定义的字段。这个列表可能 包括一个或多个字段，或由逗号分隔开的字段区间。
-d delim_char	当指定-f 选项之后，使用 delim_char 做为字段分隔符。默认情况下， 字段之间必须由单个 tab 字符分隔开。
--complement	抽取整个文本行，除了那些由-c 和／或-f 选项指定的文本。

正如我们所看到的，cut 程序抽取文本的方式相当不灵活。cut 命令最好用来从其它程序产生的文件中 抽取文本，而不是从人们直接输入的文本中抽取。我们将会看一下我们的 distros.txt 文件，看看 是否它足够 “整齐” 成为 cut 实例的一个好样本。如果我们使用带有 -A 选项的 cat 命令，我们能查看是否这个 文件符号由 tab 字符分离字段的要求。

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

看起来不错。字段之间仅仅是单个 tab 字符，没有嵌入空格。因为这个文件使用了 tab 而不是空格， 我们将使用 -f 选项来抽取一个字段：

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

因为我们的 distros 文件是由 tab 分隔开的，最好用 cut 来抽取字段而不是字符。这是因为一个由 tab 分离的文件， 每行不太可能包含相同的字符数，这就使计算每行中字符的位置变得困难或者是不可能。在以上事例中，然而， 我们已经抽取了一个字段，幸运地是其包含地日期长度相同，所以通过从每行中抽取年份，我们能展示怎样 来抽取字符：

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2007
2006
2007
2006
2008
2006
2008
2006
```

通过对我们的列表再次运行 cut 命令，我们能够抽取从位置7到10的字符，其对应于日期字段的年份。 这个 7-10 表示法是一个区间的例子。cut 命令手册包含了一个如何指定区间的完整描述。

> 展开 *Tabs*
>
> distros.txt 的文件格式很适合使用 cut 程序来抽取字段。但是如果我们想要 cut 程序 按照字符，而不是字段来操作一个文件，那又怎样呢？这要求我们用相应数目的空格来 代替 tab 字符。幸运地是，GNU 的 Coreutils 软件包有一个工具来解决这个问题。这个 程序名为 expand，它既可以接受一个或多个文件参数，也可以接受标准输入，并且把 修改过的文本送到标准输出。
>
> 如果我们通过 expand 来处理 distros.txt 文件，我们能够使用 cut -c 命令来从文件中抽取 任意区间内的字符。例如，我们能够使用以下命令来从列表中抽取发行年份，通过展开 此文件，再使用 cut 命令，来抽取从位置 23 开始到行尾的每一个字符：
>
> *[me@linuxbox ~]$ expand distros.txt | cut -c 23-*
>
> Coreutils 软件包也提供了 unexpand 程序，用 tab 来代替空格。

当操作字段的时候，有可能指定不同的字段分隔符，而不是 tab 字符。这里我们将会从/etc/passwd 文件中 抽取第一个字段：

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

使用-d 选项，我们能够指定冒号做为字段分隔符。

这个 paste 命令的功能正好与 cut 相反。它会添加一个或多个文本列到文件中，而不是从文件中抽取文本列。 它通过读取多个文件，然后把每个文件中的字段整合成单个文本流，输入到标准输出。类似于 cut 命令， paste 接受多个文件参数和 ／ 或标准输入。为了说明 paste 是怎样工作的，我们将会对 distros.txt 文件 动手术，来产生发行版的年代表。

从我们之前使用 sort 的工作中，首先我们将产生一个按照日期排序的发行版列表，并把结果 存储在一个叫做 distros-by-date.txt 的文件中：

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > distros-by-date.txt
```

下一步，我们将会使用 cut 命令从文件中抽取前两个字段（发行版名字和版本号），并把结果存储到 一个名为 distro-versions.txt 的文件中：

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.txt
[me@linuxbox ~]$ head distros-versions.txt
Fedora     10
Ubuntu     8.10
SUSE       11.0
Fedora     9
Ubuntu     8.04
Fedora     8
Ubuntu     7.10
SUSE       10.3
Fedora     7
Ubuntu     7.04
```

最后的准备步骤是抽取发行日期，并把它们存储到一个名为 distro-dates.txt 文件中：

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

现在我们拥有了我们所需要的文本了。为了完成这个过程，使用 paste 命令来把日期列放到发行版名字 和版本号的前面，这样就创建了一个年代列表。通过使用 paste 命令，然后按照期望的顺序来安排它的 参数，就能很容易完成这个任务。

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008    Fedora     10
10/30/2008    Ubuntu     8.10
06/19/2008    SUSE       11.0
05/13/2008    Fedora     9
04/24/2008    Ubuntu     8.04
11/08/2007    Fedora     8
10/18/2007    Ubuntu     7.10
10/04/2007    SUSE       10.3
05/31/2007    Fedora     7
04/19/2007    Ubuntu     7.04
```

在某些方面，join 命令类似于 paste，它会往文件中添加列，但是它使用了独特的方法来完成。 一个 join 操作通常与关系型数据库有关联，在关系型数据库中来自多个享有共同关键域的表格的 数据结合起来，得到一个期望的结果。这个 join 程序执行相同的操作。它把来自于多个基于共享 关键域的文件的数据结合起来。

为了知道在关系数据库中是怎样使用 join 操作的，让我们想象一个很小的数据库，这个数据库由两个 表格组成，每个表格包含一条记录。第一个表格，叫做 CUSTOMERS，有三个数据域：一个客户号（CUSTNUM）， 客户的名字（FNAME）和客户的姓（LNAME）：

```
CUSTNUM        FNAME       ME
========    =====      ======
4681934        John        Smith
```

第二个表格叫做 ORDERS，其包含四个数据域：订单号（ORDERNUM），客户号（CUSTNUM），数量（QUAN）， 和订购的货品（ITEM）。

```
ORDERNUM        CUSTNUM     QUAN ITEM
========       =======     ==== ====
3014953305      4681934     1    Blue Widget
```

注意两个表格共享数据域 CUSTNUM。这很重要，因为它使表格之间建立了联系。

执行一个 join 操作将允许我们把两个表格中的数据域结合起来，得到一个有用的结果，例如准备 一张发货单。通过使用两个表格 CUSTNUM 数字域中匹配的数值，一个 join 操作会产生以下结果：

```
FNAME       LNAME       QUAN ITEM
=====       =====       ==== ====
John        Smith       1    Blue Widget
```

为了说明 join 程序，我们需要创建一对包含共享键值的文件。为此，我们将使用我们的 distros.txt 文件。 从这个文件中，我们将构建额外两个文件，一个包含发行日期（其会成为共享键值）和发行版名称：

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008 Fedora
10/30/2008 Ubuntu
06/19/2008 SUSE
05/13/2008 Fedora
04/24/2008 Ubuntu
11/08/2007 Fedora
10/18/2007 Ubuntu
10/04/2007 SUSE
05/31/2007 Fedora
04/19/2007 Ubuntu
```

第二个文件包含发行日期和版本号：

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > distros-key-vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008 10
10/30/2008 8.10
06/19/2008 11.0
05/13/2008 9
04/24/2008 8.04
11/08/2007 8
10/18/2007 7.10
10/04/2007 10.3
05/31/2007 7
04/19/2007 7.04
```

现在我们有两个具有共享键值（“发行日期” 数据域）的文件。有必要指出，为了使 join 命令 能正常工作，所有文件必须按照关键数据域排序。

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt | head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

也要注意，默认情况下，join 命令使用空白字符做为输入字段的界定符，一个空格作为输出字段 的界定符。这种行为可以通过指定的选项来修改。详细信息，参考 join 命令手册。

比较文本

通常比较文本文件的版本很有帮助。对于系统管理员和软件开发者来说，这个尤为重要。 一名系统管理员可能，例如，需要拿现有的配置文件与先前的版本做比较，来诊断一个系统错误。 同样的，一名程序员经常需要查看程序的修改。

这个 comm 程序会比较两个文本文件，并且会显示每个文件特有的文本行和共有的文把行。 为了说明问题，通过使用 cat 命令，我们将会创建两个内容几乎相同的文本文件：

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

下一步，我们将使用 comm 命令来比较这两个文件：

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
        b
        c
        d
    e
```

正如我们所见到的，comm 命令产生了三列输出。第一列包含第一个文件独有的文本行；第二列， 文本行是第二列独有的；第三列包含两个文件共有的文本行。comm 支持 -n 形式的选项，这里 n 代表 1，2 或 3。这些选项使用的时候，指定了要隐藏的列。例如，如果我们只想输出两个文件共享的文本行， 我们将隐藏第一列和第二列的输出结果：

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

类似于 comm 程序，diff 程序被用来监测文件之间的差异。然而，diff 是一款更加复杂的工具，它支持 许多输出格式，并且一次能处理许多文本文件。软件开发员经常使用 diff 程序来检查不同程序源码 版本之间的更改，diff 能够递归地检查源码目录，经常称之为源码树。diff 程序的一个常见用例是 创建 diff 文件或者补丁，它会被其它程序使用，例如 patch 程序（我们一会儿讨论），来把文件 从一个版本转换为另一个版本。

如果我们使用 diff 程序，来查看我们之前的文件实例：

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

我们看到 diff 程序的默认输出风格：对两个文件之间差异的简短描述。在默认格式中， 每组的更改之前都是一个更改命令，其形式为*range operation range*， 用来描述要求更改的位置和类型，从而把第一个文件转变为第二个文件：

表21-4: diff 更改命令

改变	说明
r1ar2	把第二个文件中位置 r2 处的文件行添加到第一个文件中的 r1 处。
r1cr2	用第二个文件中位置 r2 处的文本行更改（替代）位置 r1 处的文本行。
r1dr2	删除第一个文件中位置 r1 处的文本行，这些文本行将会出现在第二个文件中位置 r2 处。

在这种格式中，一个范围就是由逗号分隔开的开头行和结束行的列表。虽然这种格式是默认情况（主要是 为了服从 POSIX 标准且向后与传统的 Unix diff 命令兼容）， 但是它并不像其它可选格式一样被广泛地使用。最流行的两种格式是上下文模式和统一模式。

当使用上下文模式（带上 -c 选项），我们将看到这些：

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt    2008-12-23 06:40:13.000000000 -0500
--- file2.txt   2008-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b
  c
  d
  + e
```

这个输出结果以两个文件名和它们的时间戳开头。第一个文件用星号做标记，第二个文件用短横线做标记。 纵观列表的其它部分，这些标记将象征它们各自代表的文件。下一步，我们看到几组修改， 包括默认的周围上下文行数。在第一组中，我们看到：

```
*** 1,4 ***
```

其表示第一个文件中从第一行到第四行的文本行。随后我们看到：

```
--- 1,4 ---
```

这表示第二个文件中从第一行到第四行的文本行。在更改组内，文本行以四个指示符之一开头：

表21-5：diff 上下文模式更改指示符

指示符	意思
blank	上下文显示行。它并不表示两个文件之间的差异。
-	删除行。这一行将会出现在第一个文件中，而不是第二个文件内。
+	添加行。这一行将会出现在第二个文件内，而不是第一个文件中。
!	更改行。将会显示某个文本行的两个版本，每个版本会出现在更改组的各自部分。

这个统一模式相似于上下文模式，但是更加简洁。通过 -u 选项来指定它：

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt 2008-12-23 06:40:13.000000000 -0500
+++ file2.txt 2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

上下文模式和统一模式之间最显著的差异就是重复上下文的消除，这就使得统一模式的输出结果要比上下文 模式的输出结果简短。在我们上述实例中，我们看到类似于上下文模式中的文件时间戳，其紧紧跟随字符串 @@ -1,4 +1,4 @@。这行字符串表示了在更改组中描述的第一个文件中的文本行和第二个文件中的文本行。 这行字符串之后就是文本行本身，与三行默认的上下文。每行以可能的三个字符中的一个开头：

表21-6：diff 统一模式更改指示符

字符	意思
空格	两个文件都包含这一行。
-	在第一个文件中删除这一行。
+	添加这一行到第一个文件中。

这个 patch 程序被用来把更改应用到文本文件中。它接受从 diff 程序的输出，并且通常被用来 把较老的文件版本转变为较新的文件版本。让我们考虑一个著名的例子。Linux 内核是由一个 大型的，组织松散的贡献者团队开发而成，这些贡献者会提交固定的少量更改到源码包中。 这个 Linux 内核由几百万行代码组成，虽然每个贡献者每次所做的修改相当少。对于一个贡献者 来说，每做一个修改就给每个开发者发送整个的内核源码树，这是没有任何意义的。相反， 提交一个 diff 文件。一个 diff 文件包含先前的内核版本与带有贡献者修改的新版本之间的差异。 然后一个接受者使用 patch 程序，把这些更改应用到他自己的源码树中。使用 diff/patch 组合提供了 两个重大优点：

1. 一个 diff 文件非常小，与整个源码树的大小相比较而言。

2. 一个 diff 文件简洁地显示了所做的修改，从而允许程序补丁的审阅者能快速地评估它。

当然，diff/patch 能工作于任何文本文件，不仅仅是源码文件。它同样适用于配置文件或任意其它文本。

准备一个 diff 文件供 patch 程序使用，GNU 文档（查看下面的拓展阅读部分）建议这样使用 diff 命令：

```
diff -Naur old_file new_file > diff_file
```

old_file 和 new_file 部分不是单个文件就是包含文件的目录。这个 r 选项支持递归目录树。

一旦创建了 diff 文件，我们就能应用它，把旧文件修补成新文件。

```
patch < diff_file
```

我们将使用测试文件来说明：

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt &gt; patchfile.txt
[me@linuxbox ~]$ patch &lt; patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

在这个例子中，我们创建了一个名为 patchfile.txt 的 diff 文件，然后使用 patch 程序， 来应用这个补丁。注意我们没有必要指定一个要修补的目标文件，因为 diff 文件（在统一模式中）已经 在标题行中包含了文件名。一旦应用了补丁，我们能看到，现在 file1.txt 与 file2.txt 文件相匹配了。

patch 程序有大量的选项，而且还有额外的实用程序可以被用来分析和编辑补丁。

运行时编辑

我们对于文本编辑器的经验是它们主要是交互式的，意思是我们手动移动光标，然后输入我们的修改。 然而，也有非交互式的方法来编辑文本。有可能，例如，通过单个命令把一系列修改应用到多个文件中。

这个 tr 程序被用来更改字符。我们可以把它看作是一种基于字符的查找和替换操作。 换字是一种把字符从一个字母转换为另一个字母的过程。例如，把小写字母转换成大写字母就是 换字。我们可以通过 tr 命令来执行这样的转换，如下所示：

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

正如我们所见，tr 命令操作标准输入，并把结果输出到标准输出。tr 命令接受两个参数：要被转换的字符集以及 相对应的转换后的字符集。字符集可以用三种方式来表示：

1. 一个枚举列表。例如， ABCDEFGHIJKLMNOPQRSTUVWXYZ
2. 一个字符域。例如，A-Z 。注意这种方法有时候面临与其它命令相同的问题，归因于 语系的排序规则，因此应该谨慎使用。
3. POSIX 字符类。例如，[:upper:]

大多数情况下，两个字符集应该长度相同；然而，有可能第一个集合大于第二个，尤其如果我们 想要把多个字符转换为单个字符：

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

除了换字之外，tr 命令能允许字符从输入流中简单地被删除。在之前的章节中，我们讨论了转换 MS-DOS 文本文件为 Unix 风格文本的问题。为了执行这个转换，每行末尾的回车符需要被删除。 这个可以通过 tr 命令来执行，如下所示：

```
tr -d '\r' < dos_file > unix_file
```

这里的 dos_file 是需要被转换的文件，unix_file 是转换后的结果。这种形式的命令使用转义序列 \r 来代表回车符。查看 tr 命令所支持地完整的转义序列和字符类别列表，试试下面的命令：

```
[me@linuxbox ~]$ tr --help
```

> *ROT13:* 不那么秘密的编码环
>
> tr 命令的一个有趣的用法是执行 ROT13文本编码。ROT13是一款微不足道的基于一种简易的替换暗码的 加密类型。把 ROT13称为“加密”是大方的；“文本模糊处理”更准确些。有时候它被用来隐藏文本中潜在的攻击内容。 这个方法就是简单地把每个字符在字母表中向前移动13位。因为移动的位数是可能的26个字符的一半， 所以对文本再次执行这个算法，就恢复到了它最初的形式。通过 tr 命令来执行这种编码：
>
> *echo "secret text" | tr a-zA-Z n-za-mN-ZA-M*
>
> frperg grkg
>
> 再次执行相同的过程，得到翻译结果：
>
> *echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M*
>
> secret text
>
> 大量的 email 程序和 USENET 新闻读者都支持 ROT13 编码。Wikipedia 上面有一篇关于这个主题的好文章：
>
> http://en.wikipedia.org/wiki/ROT13

tr 也可以完成另一个技巧。使用-s 选项，tr 命令能“挤压”（删除）重复的字符实例：

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

这里我们有一个包含重复字符的字符串。通过给 tr 命令指定字符集“ab”，我们能够消除字符集中 字母的重复实例，然而会留下不属于字符集的字符（“c”）无更改。注意重复的字符必须是相邻的。 如果它们不相邻：

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

那么挤压会没有效果。

名字 sed 是 stream editor（流编辑器）的简称。它对文本流进行编辑，要不是一系列指定的文件， 要不就是标准输入。sed 是一款强大的，并且有些复杂的程序（有整本内容都是关于 sed 程序的书籍）， 所以在这里我们不会详尽的讨论它。

总之，sed 的工作方式是要不给出单个编辑命令（在命令行中）要不就是包含多个命令的脚本文件名， 然后它就按行来执行这些命令。这里有一个非常简单的 sed 实例：

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

在这个例子中，我们使用 echo 命令产生了一个单词的文本流，然后把它管道给 sed 命令。sed，依次， 对流文本执行指令 s/front/back/，随后输出“back”。我们也能够把这个命令认为是相似于 vi 中的“替换” （查找和替代）命令。

sed 中的命令开始于单个字符。在上面的例子中，这个替换命令由字母 s 来代表，其后跟着查找 和替代字符串，斜杠字符做为分隔符。分隔符的选择是随意的。按照惯例，经常使用斜杠字符， 但是 sed 将会接受紧随命令之后的任意字符做为分隔符。我们可以按照这种方式来执行相同的命令：

```
[me@linuxbox ~]$ echo "front" | sed 's\_front\_back\_'
back
```

通过紧跟命令之后使用下划线字符，则它变成界定符。sed 可以设置界定符的能力，使命令的可读性更强， 正如我们将看到的.

sed 中的大多数命令之前都会带有一个地址，其指定了输入流中要被编辑的文本行。如果省略了地址， 然后会对输入流的每一行执行编辑命令。最简单的地址形式是一个行号。我们能够添加一个地址 到我们例子中：

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

给我们的命令添加地址 1，就导致只对仅有一行文本的输入流的第一行执行替换操作。如果我们指定另一 个数字：

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

我们看到没有执行这个编辑命令，因为我们的输入流没有第二行。地址可以用许多方式来表达。这里是 最常用的：

表21-7: sed 地址表示法

地址	说明
n	行号，n 是一个正整数。
$	最后一行。
/regexp/	所有匹配一个 POSIX 基本正则表达式的文本行。注意正则表达式通过 斜杠字符界定。选择性地，这个正则表达式可能由一个备用字符界定，通过\cregexpc 来 指定表达式，这里 c 就是一个备用的字符。
addr1,addr2	从 addr1 到 addr2 范围内的文本行，包含地址 addr2 在内。地址可能是上述任意 单独的地址形式。
first~step	匹配由数字 first 代表的文本行，然后随后的每个在 step 间隔处的文本行。例如 1~2 是指每个位于偶数行号的文本行，5~5 则指第五行和之后每五行位置的文本行。
addr1,+n	匹配地址 addr1 和随后的 n 个文本行。
addr!	匹配所有的文本行，除了 addr 之外，addr 可能是上述任意的地址形式。

通过使用这一章中早前的 distros.txt 文件，我们将演示不同种类的地址表示法。首先，一系列行号：

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE        10.2    12/07/2006
Fedora      10      11/25/2008
SUSE        11.0    06/19/2008
Ubuntu      8.04    04/24/2008
Fedora      8       11/08/2007
```

在这个例子中，我们打印出一系列的文本行，开始于第一行，直到第五行。为此，我们使用 p 命令， 其就是简单地把匹配的文本行打印出来。然而为了高效，我们必须包含选项 -n（不自动打印选项）， 让 sed 不要默认地打印每一行。

下一步，我们将试用一下正则表达式：

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE      10.2    12/07/2006
SUSE      11.0    06/19/2008
SUSE      10.3    10/04/2007
SUSE      10.1    05/11/2006
```

通过包含由斜杠界定的正则表达式 VSUSEV，我们能够孤立出包含它的文本行，和 grep 程序的功能 是相同的。

最后，我们将试着否定上面的操作，通过给这个地址添加一个感叹号：

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora      10      11/25/2008
Ubuntu      8.04    04/24/2008
Fedora      8       11/08/2007
Ubuntu      6.10    10/26/2006
Fedora      7       05/31/2007
Ubuntu      7.10    10/18/2007
Ubuntu      7.04    04/19/2007
Fedora      6       10/24/2006
Fedora      9       05/13/2008
Ubuntu      6.06    06/01/2006
Ubuntu      8.10    10/30/2008
Fedora      5       03/20/2006
```

这里我们看到期望的结果：输出了文件中所有的文本行，除了那些匹配这个正则表达式的文本行。

目前为止，我们已经知道了两个 sed 的编辑命令，s 和 p。这里是一个更加全面的基本编辑命令列表：

表 21-8： sed 基本编辑命令

命令	说明
=	输出当前的行号。
a	在当前行之后追加文本。
d	删除当前行。
i	在当前行之前插入文本。
p	打印当前行。默认情况下，sed 程序打印每一行，并且只是编辑文件中匹配 指定地址的文本行。通过指定-n 选项，这个默认的行为能够被忽略。
q	退出 sed，不再处理更多的文本行。如果不指定-n 选项，输出当前行。
Q	退出 sed，不再处理更多的文本行。
s/regexp/replacement/	只要找到一个 regexp 匹配项，就替换为 replacement 的内容。 replacement 可能包括特殊字符 &，其等价于由 regexp 匹配的文本。另外， replacement 可能包含序列 \1 到 \9，其是 regexp 中相对应的子表达式的内容。更多信息，查看 下面 back references 部分的讨论。在 replacement 末尾的斜杠之后，可以指定一个 可选的标志，来修改 s 命令的行为。
y/set1/set2	执行字符转写操作，通过把 set1 中的字符转变为相对应的 set2 中的字符。 注意不同于 tr 程序，sed 要求两个字符集合具有相同的长度。

到目前为止，这个 s 命令是最常使用的编辑命令。我们将仅仅演示一些它的功能，通过编辑我们的 distros.txt 文件。我们以前讨论过 distros.txt 文件中的日期字段不是“友好地计算机”模式。 文件中的日期格式是 MM/DD/YYYY，但如果格式是 YYYY-MM-DD 会更好一些（利于排序）。手动修改 日期格式不仅浪费时间而且易出错，但是有了 sed，只需一步就能完成修改：

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' distros.txt
SUSE        10.2    2006-12-07
Fedora      10      2008-11-25
SUSE        11.0    2008-06-19
Ubuntu      8.04    2008-04-24
Fedora      8       2007-11-08
SUSE        10.3    2007-10-04
Ubuntu      6.10    2006-10-26
Fedora      7       2007-05-31
Ubuntu      7.10    2007-10-18
Ubuntu      7.04    2007-04-19
SUSE        10.1    2006-05-11
Fedora      6       2006-10-24
Fedora      9       2008-05-13
Ubuntu      6.06    2006-06-01
Ubuntu      8.10    2008-10-30
Fedora      5       2006-03-20
```

哇！这个命令看起来很丑陋。但是它起作用了。仅用一步，我们就更改了文件中的日期格式。 它也是一个关于为什么有时候会开玩笑地把正则表达式称为是“只写”媒介的完美的例子。我们 能写正则表达式，但是有时候我们不能读它们。在我们恐惧地忍不住要逃离此命令之前，让我们看一下 怎样来构建它。首先，我们知道此命令有这样一个基本的结构：

```
sed 's/regexp/replacement/' distros.txt
```

我们下一步是要弄明白一个正则表达式将要孤立出日期。因为日期是 MM/DD/YYYY 格式，并且 出现在文本行的末尾，我们可以使用这样的表达式：

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

此表达式匹配两位数字，一个斜杠，两位数字，一个斜杠，四位数字，以及行尾。如此关心 regexp， 那么 replacement 又怎样呢？为了解决此问题，我们必须介绍一个正则表达式的新功能，它出现 在一些使用 BRE 的应用程序中。这个功能叫做逆参照，像这样工作：如果序列 \n 出现在 replacement 中 ， 这里 n 是指从 1 到 9 的数字，则这个序列指的是在前面正则表达式中相对应的子表达式。为了 创建这个子表达式，我们简单地把它们用圆括号括起来，像这样：

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

现在我们有了三个子表达式。第一个表达式包含月份，第二个包含某月中的某天，以及第三个包含年份。 现在我们就可以构建 replacement ，如下所示：

```
\3-\1-\2
```

此表达式给出了年份，一个斜杠，月份，一个斜杠，和某天。

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

我们还有两个问题。第一个是在我们表达式中额外的斜杠将会迷惑 sed，当 sed 试图解释这个 s 命令 的时候。第二个是因为 sed，默认情况下，只接受基本的正则表达式，在表达式中的几个字符会 被当作文字字面值，而不是元字符。我们能够解决这两个问题，通过反斜杠的自由应用来转义 令人不快的字符：

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' distros.txt
```

你掌握了吧!

s 命令的另一个功能是使用可选标志，其跟随替代字符串。一个最重要的可选标志是 g 标志，其 指示 sed 对某个文本行全范围地执行查找和替代操作，不仅仅是对第一个实例，这是默认行为。 这里有个例子：

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

我们看到虽然执行了替换操作，但是只针对第一个字母 “b” 实例，然而剩余的实例没有更改。通过添加 g 标志， 我们能够更改所有的实例：

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

目前为止，通过命令行我们只让 sed 执行单个命令。使用-f 选项，也有可能在一个脚本文件中构建更加复杂的命令。 为了演

示，我们将使用 sed 和 distros.txt 文件来生成一个报告。我们的报告以开头标题，修改过的日期，以及 大写的发行版名称为特征。为此，我们需要编写一个脚本，所以我们将打开文本编辑器，然后输入以下文字：

```
# sed script to produce Linux distributions report
1 i\
\
Linux Distributions Report\
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

我们将把 sed 脚本保存为 distros.sed 文件，然后像这样运行它：

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt
Linux Distributions Report
SUSE    10.2    2006-12-07
FEDORA    10        2008-11-25
SUSE    11.0    2008-06-19
UBUNTU    8.04    2008-04-24
FEDORA    8        2007-11-08
SUSE    10.3    2007-10-04
UBUNTU    6.10    2006-10-26
FEDORA    7        2007-05-31
UBUNTU    7.10    2007-10-18
UBUNTU    7.04    2007-04-19
SUSE    10.1    2006-05-11
FEDORA    6        2006-10-24
FEDORA    9        2008-05-13
```

正如我们所见，我们的脚本文件产生了期望的结果，但是它是如何做到的呢？让我们再看一下我们的脚本文件。 我们将使用 cat 来给每行文本编号：

```
[me@linuxbox ~]$ cat -n distros.sed
1 # sed script to produce Linux distributions report
2
3 1 i\
4 \
5 Linux Distributions Report\
6
7 s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
8 y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

我们脚本文件的第一行是一条注释。如同 Linux 系统中的许多配置文件和编程语言一样，注释以#字符开始， 然后是人类可读的文本。注释可以被放到脚本中的任意地方（虽然不在命令本身之中），且对任何 可能需要理解和／或维护脚本的人们都很有帮助。

第二行是一个空行。正如注释一样，添加空白行是为了提高程序的可读性。

许多 sed 命令支持行地址。这些行地址被用来指定对输入文本的哪一行执行操作。行地址可能被 表示为单独的行号，行号范围，以及特殊的行号“$”，它表示输入文本的最后一行。

从第三行到第六行所包含地文本要被插入到地址 1 处，也就是输入文本的第一行中。这个 i 命令 之后是反斜杠回车符，来产生一个转义的回车符，或者就是所谓的连行符。这个序列能够 被用在许多环境下，包括 shell 脚本，从而允许把回车符嵌入到文本流中，而没有通知 解释器（在这是指 sed 解释器）已经到达了文本行的末尾。这个 i 命令，同样地，命令 a（追加文本， 而不是插入文本）和 c（取代文本）命令都允许多个文本行，只要每个文本行，除了最后一行，以一个 连行符结束。实际上，脚本的第六行是插入文本的末尾，它以一个普通的回车符结尾，而不是一个 连行符，通知解释器 i 命令结束了。

注意：一个连行符由一个斜杠字符其后紧跟一个回车符组成。它们之间不允许有空白字符。

第七行是我们的查找和替代命令。因为命令之前没有添加地址，所以输入流中的每一行文本 都得服从它的操作。

第八行执行小写字母到大写字母的字符替换操作。注意不同于 tr 命令，这个 sed 中的 y 命令不 支持字符区域（例如，[a-z]），也不支持 POSIX 字符集。再说一次，因为 y 命令之前不带地址， 所以它会操作输入流的每一行。

> 喜欢 *sed* 的人们也会喜欢。。。
>
> sed 是一款非常强大的程序，它能够针对文本流完成相当复杂的编辑任务。它最常 用于简单的行任务，而不是长长的脚本。许多用户喜欢使用其它工具，来执行较大的工作。 在这些工具中最著名的是 awk 和 perl。它们不仅仅是工具，像这里介绍的程序，且延伸到 完整的编程语言领域。特别是 perl，经常被用来代替 shell 脚本，来完成许多系统管理任务， 同时它也是一款非常流行网络开发语言。awk 更专用一些。其具体优点是其操作表格数据的能力。 awk 程序通常逐行处理文本文件，这点类似于 sed，awk 使用了一种方案，其与 sed 中地址 之后跟随编辑命令的概念相似。虽然关于 awk 和 perl 的内容都超出了本书所讨论的范围， 但是对于 Linux 命令行用户来说，它们都是非常好的技能。

我们要查看的最后一个工具是 aspell，一款交互式的拼写检查器。这个 aspell 程序是早先 ispell 程序 的继承者，大多数情况下，它可以被用做一个替代品。虽然 aspell 程序大多被其它需要拼写检查能力的 程序使用，但它也可以作为一个独立的命令行工具使用。它能够智能地检查各种类型的文本文件， 包括 HTML 文件，C/C++ 程序，电子邮件和其它种类的专业文本。

拼写检查一个包含简单的文本文件，可以这样使用 aspell:

```
aspell check textfile
```

这里的 textfile 是要检查的文件名。作为一个实际例子，让我们创建一个简单的文本文件，叫做 foo.txt， 包含一些故意的拼写错误：

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

下一步我们将使用 aspell 来检查文件：

```
[me@linuxbox ~]$ aspell check foo.txt
```

因为 aspell 在检查模式下是交互的，我们将看到像这样的一个屏幕：

```
The quick brown fox jimped over the laxy dog.
1)jumped                6)wimped
2)gimped                7)camped
3)comped                8)humped
4)limped                9)impede
5)pimped                0)umped
i)Ignore                I)Ignore all
r)Replace               R)Replace all
a)Add                   l)Add Lower
b)Abort                 x)Exit
?
```

在显示屏的顶部，我们看到我们的文本中有一个拼写可疑且高亮显示的单词。在中间部分，我们看到 十个拼写建议，序号从 0 到 9，然后是一系列其它可能的操作。最后，在最底部，我们看到一个提示符， 准备接受我们的选择。

如果我们按下 1 按键，aspell 会用单词 jumped 代替错误单词，然后移动到下一个拼写错的单词，就是 laxy。如果我们选择替代物 lazy，aspell 会替换 laxy 并且终止。一旦 aspell 结束操作，我们 可以检查我们的文件，会看到拼写错误的单词已经更正了。

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

除非由命令行选项 --dont-backup 告诉 aspell，否则通过追加扩展名 .bak 到文件名中, aspell 会创建一个包含原始文本的备

份文件。

为了炫耀 sed 的编辑本领，我们将还原拼写错误，从而能够重用我们的文件：

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

这个 sed 选项 -i，告诉 sed 在适当位置编辑文件，意思是不要把编辑结果发送到标准输出中。sed 会把更改应用到文件中，以此重新编写文件。我们也看到可以把多个 sed 编辑命令放在同一行，编辑命令之间由分号分隔开来。

下一步，我们将看一下 aspell 怎样来解决不同种类的文本文件。使用一个文本编辑器，例如 vim（胆大的人可能想用 sed）， 我们将添加一些 HTML 标志到文件中：

```
<html>
   <head>
        <title>Mispelled HTML file</title>
   </head>
   <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
   </body>
</html>
```

现在，如果我们试图拼写检查我们修改的文件，我们会遇到一个问题。如果我们这样做：

```
[me@linuxbox ~]$ aspell check foo.txt
```

我们会得到这些：

```
<html>
   <head>
        <title>Mispelled HTML file</title>
   </head>
   <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
   </body>
</html>
1) HTML                 4) Hamel
2) ht ml                5) Hamil
3) ht-ml                6) hotel

i) Ignore               I) Ignore all
r) Replace              R) Replace all
a) Add                  l) Add Lower
b) Abort                x) Exit
?
```

aspell 会认为 HTML 标志的内容是拼写错误。通过包含 -H（HTML）检查模式选项，这个问题能够 解决，像这样：

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

这会导致这样的结果：

```
<html>
  <head>
      <title><b>Mispelled</b> HTML file</title>
  </head>
  <body>
      <p>The quick brown fox jimped over the laxy dog.</p>
  </body>
</html>
1) HTML                  4) Hamel
2) ht ml                 5) Hamil
3) ht-ml                 6) hotel
i) Ignore                I) Ignore all
r) Replace               R) Replace all
a) Add                   l) Add Lower
b) Abort                 x) Exit
?
```

这个 HTML 标志被忽略了，并且只会检查文件中非标志部分的内容。在这种模式下，HTML 标志的 内容被忽略了，不会进行拼写检查。然而，ALT 标志的内容，会被检查。

注意：默认情况下，aspell 会忽略文本中 URL 和电子邮件地址。通过命令行选项，可以重写此行为。 也有可能指定哪些标志进行检查及跳过。详细内容查看 aspell 命令手册。

总结归纳

在这一章中，我们已经查看了一些操作文本的命令行工具。在下一章中，我们会再看几个命令行工具。 诚然，看起来不能立即显现出怎样或为什么你可能使用这些工具为日常的基本工具， 虽然我们已经展示了一些半实际的命令用法的例子。我们将在随后的章节中发现这些工具组成 了解决实际问题的基本工具箱。这将是确定无疑的，当我们学习 shell 脚本的时候， 到时候这些工具将真正体现出它们的价值。

拓展阅读

GNU 项目网站包含了本章中所讨论工具的许多在线指南。

- 来自 Coreutils 软件包：

  http://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files

  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files

  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields-within-a-line

  http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-characters

- 来自 Diffutils 软件包：

  http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

- sed

  http://www.gnu.org/software/sed/manual/sed.html

- aspell

  http://aspell.net/man-html/index.html

- There are many other online resources for sed, in particular:

  http://www.grymoire.com/Unix/Sed.html

http://sed.sourceforge.net/sed1line.txt

- 也试着搜索一下 “sed one liners”, “sed cheat sheets”

友情提示

有一些更有趣的文本操作命令值得。在它们之间有：split（把文件分割成碎片）， csplit（基于上下文把文件分割成碎片），和 sdiff（并排合并文件差异）。

格式化输出

在这章中，我们继续着手于文本相关的工具，关注那些用来格式化输出的程序，而不是改变文本自身。 这些工具通常让文本准备就绪打印，这是我们在下一章会提到的。我们在这章中会提到的工具有：

- nl – 添加行号
- fold – 限制文件列宽
- fmt – 一个简单的文本格式转换器
- pr – 让文本为打印做好准备
- printf – 格式化数据并打印出来
- groff – 一个文件格式系统

简单的格式化工具

我们将先着眼于一些简单的格式工具。他们都是功能单一的程序，并且做法有一点单纯，但是他们能被用于小任务并且作为脚本和管道的一部分。

nl – 添加行号

nl 程序是一个相当神秘的工具，用作一个简单的任务。它添加文件的行数。在它最简单的用途中，它相当于 cat -n:

```
[me@linuxbox ~]$ nl distros.txt | head
```

像 cat，nl 既能接受多个文件作为命令行参数，也能标准输出。然而，nl 有一个相当数量的选项并支持一个简单的标记方式去允许更多复杂的方式的计算。

nl 在计算文件行数的时候支持一个叫“逻辑页面”的概念。这允许 nl 在计算的时候去重设（再一次开始）可数的序列。用到那些选项 的时候，可以设置一个特殊的开始值， 并且在某个可限定的程度上还能设置它的格式。一个逻辑页面被进一步分为 header, body 和 footer 这样的元素。在每一个部分中，数行数可以被重设，并且/或被设置成另外一个格式。如果nl同时处理多个文件，它会把他们当成一个单一的 文本流。文本流中的部分被一些相当古怪的标记的存在加进了文本：

每一个上述的标记元素肯定在自己的行中独自出现。在处理完一个标记元素之后，nl 把它从文本流中删除。

这里有一些常用的 nl 选项：

表格 22-2: 常用 nl 选项

选项	含义
-b style	把 body 按被要求方式数行，可以是以下方式： a = 数所有行 t = 数非空行。这是默认设置。 n = 无 pregexp = 只数那些匹配了正则表达式的行
-f style	将 footer 按被要求设置数。默认是无
-h style	将 header 按被要求设置数。默认是
-i number	将页面增加量设置为数字。默认是一。
	设置数数的格式，格式可以是： ln = 左偏，没有前导零。

-n format	rn = 右偏，没有前导零。 rz = 右偏，有前导零。
-p	不要在没一个逻辑页面的开始重设页面数。
-s string	在没一个行的末尾加字符作分割符号。默认是单个的 tab。
-v number	将每一个逻辑页面的第一行设置成数字。默认是一。
-w width	将行数的宽度设置，默认是六。

坦诚的说，我们大概不会那么频繁地去数行数，但是我们能用 nl 去查看我们怎么将多个工具结合在一个去完成更复杂的任务。 我们将在之前章节的基础上做一个 Linux 发行版的报告。因为我们将使用 nl，包含它的 header/body/footer 标记将会十分有用。 我们将把它加到上一章的 sed 脚本来做这个。使用我们的文本编辑器，我们将脚本改成一下并且把它保存成 distros-nl.sed:

```
# sed script to produce Linux distributions report
1 i\
\\:\\:\\:\
\
Linux Distributions Report\
\
Name
Ver. Released\
----
---- --------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ i\
\\:\
\
End Of Report
```

这个脚本现在加入了 nl 的逻辑页面标记并且在报告的最后加了一个 footer。记得我们在我们的标记中必须两次使用反斜杠，因为他们通常被 sed 解释成一个转义字符。

下一步， 我们将结合 sort, sed, nl 来生成我们改进的报告：

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.sed | nl
       Linux Distributions Report
       Name    Ver.   Released
       ----    ----   --------
   1   Fedora  5      2006-03-20
   2   Fedora  6      2006-10-24
   3   Fedora  7      2007-05-31
   4   Fedora  8      2007-11-08
   5   Fedora  9      2008-05-13
   6   Fedora  10     2008-11-25
   7   SUSE    10.1   2006-05-11
   8   SUSE    10.2   2006-12-07
   9   SUSE    10.3   2007-10-04
   10  SUSE    11.0   2008-06-19
   11  Ubuntu  6.06   2006-06-01
   12  Ubuntu  6.10   2006-10-26
   13  Ubuntu  7.04   2007-04-19
   14  Ubuntu  7.10   2007-10-18
   15  Ubuntu  8.04   2008-04-24
       End Of Report
```

我们的报告是一串命令的结果，首先，我们给名单按发行版本和版本号（表格1和2处）进行排序，然后我们用 sed 生产结果， 增加了 header（包括了为 nl 增加的逻辑页面标记）和 footer。最后，我们按默认用 nl 生成了结果，只数了属于逻辑页面的 body 部分的 文本流的行数。

我们能够重复命令并且实验不同的 nl 选项。一些有趣的方式：

```
nl -n rz
```

和

```
nl -w 3 -s ' '
```

fold – 限制文件列宽

折叠是将文本的行限制到特定的宽的过程。像我们的其他命令，fold 接受一个或多个文件及标准输入。如果我们将 一个简单的文本流 fold，我们可以看到它工具的方式：

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog."
| fold -w 12
The quick br
own fox jump
ed over the
lazy dog.
```

这里我们看到了 fold 的行为 。这个用 echo 命令发送的文本用 -w 选项分解成块。在这个例子中，我们设定了行宽为12个字符。 如果没有字符设置，默认是80。注意到文本行不会因为单词边界而不会被分解。增加的 -s 选项将让 fold 分解到最后可用的空白 字符， 即会考虑单词边界。

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog."
| fold -w 12 -s
The quick
brown fox
jumped over
the lazy
dog.
```

fmt – 一个简单的文本格式器

fmt程序同样折叠文本，外加很多功能它接受文本或标准输入并且在文本流上呈现照片转换。基础来说，他填补并且将文本粘帖在 一起并且保留了空白符和缩进。

为了解释，我们将需要一些文本。让我们抄一些 fmt 主页上的东西吧：

我们将把这段文本复制进我们的文本编辑器并且保存文件名为 fmt-info.txt。现在，让我们重新格式这个文本并且让它成为一个50 个字符宽的项目。我们能用 -w 选项对文件进行处理：

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
'fmt' reads from the specified FILE arguments
(or standard input if
none are given), and writes to standard output.
By default, blank lines, spaces between words,
and indentation are
preserved in the output; successive input lines
with different indentation are not joined; tabs
are expanded on input and introduced on output.
```

好，这真是一个奇怪的结果。大概我们应该认真的阅读这段文本，因为它恰好解释了发生了什么：

默认来说，空白行，单词间距，还有缩进都会在输出中保留；持续输入不同的缩进的流不会被结合；tabs被用来扩展 输入并且引入输出 。

所以，fmt 保留了第一行的缩进。幸运的是，fmt 提供一个修正这个的选项：

好多了。通过加了 -c 选项，我们现在有了我们想要的结果。

fmt 有一些有趣的选项：

-p 选项特别有趣。通过它，我们可以格式文件选中的部分，通过在开头使用一样的符号。很多编程语言使用锚标记（#）去提醒注释的开始，而且它可以 通过这个选项来被格式。让我们创建一个有用到注释的程序。

```
[me@linuxbox ~]$ cat > fmt-code.txt
# This file contains code with comments.
# This line is a comment.
# Followed by another comment line.
# And another.
This, on the other hand, is a line of code.
And another line of code.
And another.
```

我们的示例文件包含了用 “#” 开始的注释（一个 # 后跟着一个空白符）和代码。现在，使用 fmt，我们能格式注释并且 不让代码被触及。

打印

前几章我们学习了如何操控文本，下面要做的是将文本呈于纸上。在这章中，我们将会着手用于打印文件和控制打印选项的命令行工具。通常不同发行版的打印配置各有不同且都会在其安装时自动完成，因此这里我们不讨论打印的配置过程。本章的练习需要一台正确配置的打印机来完成。 我们将讨论一下命令：

- pr —— 转换需要打印的文本文件
- lpr —— 打印文件
- lp —— 打印文件（System V）
- a2ps —— 为 PostScript 打印机格式化文件
- lpstat —— 显示打印机状态信息
- lpq —— 显示打印机队列状态
- lprm —— 取消打印任务
- cancel —— 取消打印任务（System V）

打印简史

为了较好的理解类 Unix 操作系统中的打印功能，我们必须先了解一些历史。类 Unix 系统中的打印可追溯到操作系统本身的起源，那时候打印机和它的用法与今天截然不同。

早期的打印

和计算机一样，前 PC 时代的打印机都很大、很贵，并且很集中。1980年的计算机用户都是在离电脑很远的地方用一个连接电脑的终端来工作的，而打印机就放在电脑旁并受到计算机管理员的全方位监视。

由于当时打印机既昂贵又集中，而且都工作在早期的 Unix 环境下，人们从实际考虑通常都会多人共享一台打印机。为了区别不同用户的打印任务，每个打印任务的开头都会打印一张写着用户名字的标题页，然后计算机工作人员会用推车装好当天的打印任务并分发给每个用户。

基于字符的打印机

80年代的打印机技术有两方面的不同。首先，那时的打印机基本上都是打击式打印机。打击式打印机使用撞针打击色带的机械结构在纸上形成字符。这种流行的技术造就了当时的菊轮式打印和点阵式打印。

其次，更重要的是，早期打印机的特点是它使用设备内部固定的一组字符集。比如，一台菊轮式打印机只能打印固定在其菊花轮花瓣上的字符，就这点而言打印机更像是高速打字机。大部分打字机都使用等宽字体，意思是说每个字符的宽度相等，页面上只有固定的区域可供打印，而这些区域只能容纳固定的字符数。大部分打印机采用横向10字符每英寸（CPI）和纵向6行每英寸（LPI）的规格打印，这样一张美式信片纸就有横向85字符宽纵向66行高，加上两侧的页边距，一行的最大宽度可达80字符。据此，使用等宽字体就能提供所见即所得（WYSIWYG，What You See Is What You Get）的打印预览。

接着，一台类打字机的打印机会收到以简单字节流的形式传送来的数据，其中就包含要打印的字符。例如要打印一个字母a，计算机就会发送 ASCII 码97，如果要移动打印机的滑动架和纸张，就需要使用回车、换行、换页等的小编号 ASCII 控制码。使用控制码，还能实现一些之前受限制的字体效果，比如粗体，就是让打印机先打印一个字符，然后退格再打印一遍来得到颜色较深的效果的。用 nroff 来产生一个手册页然后用 cat -A 检查输出，我们就能亲眼看看这种效果了：

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat -A | head
LS(1) User Commands LS(1)
$
$
$
N^HNA^HAM^HME^HE$
ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
l^Hls^Hs [_^HO_^HP_^HT_^HI_^HO_^HN]... [_^HF_^HI_^HL_^HE]...$
```

^H（ctrl-H）字符是用于打印粗体效果的退格符。同样，我们还可以看到用于打印下划线效果的[退格/下划线]序列。

图形化打印机

图形用户界面（GUI）的发展催生了打印机技术中主要的变革。随着计算机的展现步入更多以图形为基础的方式，打印技术也从基于字符走向图形化技术，这一切都是源于激光打印机的到来，它不仅廉价，还可以在打印区域的任意位置打印微小的墨点，而不是使用固定的字符集。这让打印机能够打印成比例的字体（像用排字机那样），甚至是图片和高质量图表。 然而，从基于字符的方式到转移到图形化的方式提出了一个严峻的技术挑战。原因如下：使用基于字符的打印机时，填满一张纸所用的字节数可以这样计算出来（假设一张纸有60行，每行80个字符）：60 × 80 = 4800 字节 相比之下，用一台300点每英寸（DPI）分辨率的激光打印机（假设一张纸有8乘10英寸的打印区域）打印则需要 (8 × 300) × (10 × 300) ÷ 8 = 900,000 字节。 当时许多慢速的个人电脑网络无法接受激光打印机打印一页需要传输将近1兆的数据这一点，因此，很有必要发明一种更聪明的方法。 这种发明便是页面描述语言（PDL）。PDL 是一种描述页面内容的编程语言。简单的说就是，“到这个地方，印一个10点大小的黑体字符 a ，到这个地方……” 这样直到页面上的所有内容都描述完了。第一种主要的 PDL 是 Adobe 系统开发的 PostScript，直到今天，这种语言仍被广泛使用。PostScript 是专为印刷各类图形和图像设计的完整的编程语言，它内建支持35种标准的高质量字体，在工作是还能够接受其他的字体定义。最早，对 PostScript 的支持是打印机本身内建的。这样传输数据的问题就解决了。相比基于字符打印机的简单字节流，典型的 PostScript 程序更为详细，而且比表示整个页面的字节数要小很多。 一台 PostScript 打印机接受 PostScript 程序作为输入。打印机有自己的处理器和内存（通常这让打印机比连接它的计算机更为强大），能执行一种叫做 PostScript 解析器的特殊程序用于读取输入的 PostScript 程序并生成结果导入打印机的内存，这样就形成了要转移到纸上的位（点）图。这种将页面渲染成大型位图（bitmap）的过程有个通用名称作光栅图像处理器（raster image processor），又叫 RIP。 多年之后，电脑和网络都变得更快了。这使得 RIP 技术从打印机转移到了主机上，还让高品质打印机变得更便宜了。 现在的许多打印机仍能接受基于字符的字节流，但很多廉价的打印机却不支持，因为它们依赖于主机的 RIP 提供的比特流来作为点阵打印。当然也有不少仍旧是 PostScript 打印机。

在 **Linux** 下打印

当前 Linux 系统采用两套软件配合显示和管理打印。第一，CUPS（Common Unix Printing System，一般 Unix 打印系统），用于提供打印驱动和打印任务管理；第二，Ghostscript，一种 PostScript 解析器，作为 RIP 使用。 CUPS 通过创建并维护打印队列来管理打印机。如前所述，Unix 下的打印原本是设计成多用户共享中央打印机的管理模式的。由于打印机本身比连接到它的电脑要慢，打印系统就需要对打印任务进行调度使其保持顺序。CUPS 还能识别出不同类型的数据（在合理范围内）并转换文件为可打印的格式。

为打印准备文件

作为命令行用户，尽管打印各种格式的文本都能实现，不过打印最多的，还是文本。

pr —— 转换需要打印的文本文件

前面的章节我们也有提到过 pr 命令，现在我们来探讨一下这条命令结合打印使用的一些选项。我们知道，在打印的历史上，基于字符的打印机曾经用过等宽字体，致使每页只能打印固定的行数和字符数，而 pr 命令则能够根据不同的页眉和页边距排列文本使其适应指定的纸张。表22-1 总结了最常用的选项。

表22－1：常用 pr 选项

选项	描述
+first[:last]	输出从 first 到 last（默认为最后）范围内的页面。
-columns	根据 columns 指定的列数排版页面内容。
-a	默认多列输出为垂直，用 -a (across) 可使其水平输出。

-d	双空格输出。
-D format	用 format 指定的格式修改页眉中显示的日期，日期命令中 format 字符串的描述详见参考手册。
-f	改用换页替换默认的回车来分割页面。
-h header	在页眉中部用 header 参数替换打印文件的名字。
-l length	设置页长为 length，默认为66行（每英寸6行的美国信纸）。
-n	输出行号。
-o offset	创建一个宽 offset 字符的左页边。
-w width	设置页宽为 width，默认为72字符。

我们通常用管道配合 pr 命令来做筛选。下面的例子中我们会列出目录 /usr/bin 并用 pr 将其格式化为3列输出的标题页：

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head
2012-02-18 14:00                    Page 1
[                 apturl          bsd-write
411toppm          ar              bsh
a2p               arecord         btcflash
a2ps              arecordmidi     bug-buddy
a2ps-lpr-wrapper  ark             buildhash
```

将打印任务送至打印机

CUPS 打印体系支持两种曾用于类 Unix 系统的打印方式。一种，叫 Berkeley 或 LPD（用于 Unix 的 Berkeley 软件发行版），使用 lpr 程序；另一种，叫 SysV（源自 System V 版本的 Unix），使用 lp 程序。这两个程序的功能大致相同。具体使用哪个完全根据个人喜好。

lpr——打印文件（Berkeley 风格）

lpr 程序可以用来把文件传送给打印机。由于它能接收标准输入，所以能用管道来协同工作。例如，要打印刚才多列目录列表的结果，我们只需这样：

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

报告会送到系统默认的打印机，如果要送到别的打印机，可以使用 -P 参数： lpr -P printer_name printer_name 表示这台打印机的名称。若要查看系统已知的打印机列表：

```
[me@linuxbox ~]$ lpstat -a
```

注意：许多 Linux 发行版允许你定义一个输出 PDF 文件但不执行实体打印的“打印机”，这可以用来很方便的检验你的打印命令。看看你的打印机配置程序是否支持这项配置。在某些发行版中，你可能要自己安装额外的软件包（如 cups-pdf）来使用这项功能。

表22-2 显示了 lpr 的一些常用选项

表22－2：常用 lpr 选项

选项	描述
-# number	设定打印份数为 number。
-p	使每页页眉标题中带有日期、时间、工作名称和页码。这种所谓的“美化打印”选项可用于打印文本文件。
-P printer	指定输出打印机的名称。未指定则使用系统默认打印机。

-r	打印后删除文件。对程序产生的临时打印文件较为有用。

lp——打印文件（System V 风格）

和 lpr 一样，lp 可以接收文件或标准输入为打印内容。与 lpr 不同的是 lp 支持不同的选项（略为复杂），表22-3 列出了其常用选项。

表22－3：常用 lp 选项

选项	描述
-d printer	设定目标（打印机）为 printer。若d选项未指定，则使用系统默认打印机。
-n number	设定的打印份数为 number。
-o landscape	设置输出为横向。
-o fitplot	缩放文件以适应页面。打印图像时较为有用，如 JPEG 文件。
-o scaling=number	缩放文件至 number。100表示填满页面，小于100表示缩小，大于100则会打印在多页上。
-o cpi=number	设定输出为 number 字符每英寸，默认为10。
-o lpi=number	设定输出为 number 行每英寸，默认为6。
-o page-bottom=points -o page-left=points -o page-right=points -o page-top=points	设置页边距，单位为点，一种印刷上的单位。一英寸 =72 点。
-P pages	指定打印的页面。pages 可以是逗号分隔的列表或范围 — 例如 1,3,5,7-10。

再次打印我们的目录列表，这次我们设置12 CPI、8 LPI 和一个半英寸的左边距。注意这里我必须调整 pr 选项来适应新的页面大小：

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36 -o cpi=12 -o lpi=8
```

这条命令用小于默认的格式产生了一个四列的列表。增加 CPI 可以让我们在页面上打印更多列。

另一种选择：a2ps

a2ps 程序很有趣。单从名字上看，这是个格式转换程序，但它的功能不止于此。程序名字的本意为 ASCII to PostScript，它是用来为 PostScript 打印机准备要打印的文本文件的。多年后，程序的功能得到了提升，名字的含义也变成了 Anything to PostScript。尽管名为格式转换程序，但它实际的功能却是打印。它的默认输出不是标准输出，而是系统的默认打印机。程序的默认行为被称为“漂亮的打印机”，这意味着它可以改善输出的外观。我们能用程序在桌面上创建一个 PostScript 文件：

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file `/home/me/Desktop/ls.ps'
```

这里我们用带 -t 参数（忽略页眉和页脚）的 pr 命令过滤数据流，然后用 a2ps 指定一个输出文件（-o 参数），并设定每页66 行（-L 参数）来匹配 pr 的输出分页。用合适的文件查看器查看我们的输出文件，我们就会看到图22-1 中显示的结果。

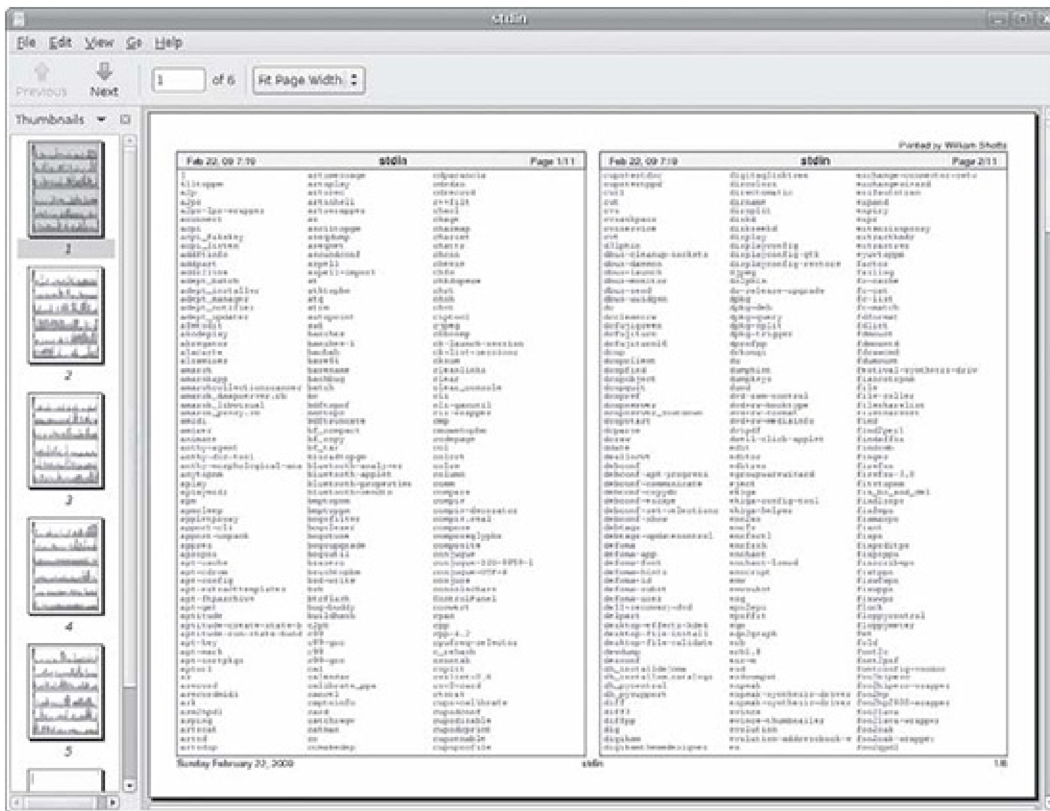


Figure 22-1: Viewing a2ps output

可以看到，默认的输出布局是一面两页的，这将导致两页的内容被打印到一张纸上。a2ps 还能利用页眉和页脚。 a2ps 有很多选项，总结在表22-4 中。

表22－4：a2ps 选项

选项	描述
--center-title text	设置中心页标题为 text。
--columns number	将所有页面排列成 number 列，默认为2。
--footer text	设置页脚为 text。
--guess	报告参数中文件的类型。由于 a2ps 会转换并格式化所有类型的数据，所以当给定文件类型后，这个选项可以很好的用来判断 a2ps 应该做什么。
--left-footer text	设置左页脚为 text。
--left-title text	设置页面左标题为 text。
--line-numbers=interval	每隔 interval 行输出行号。
--list=defauls	显示默认设置。
--list=topic	显示 topic 设置，topic 表示下列之一：代理程序（用来转换数据的外部程序），编码，特征，变量，媒介（页面大小等），ppd（PostScript 打印机描述信息），打印机，起始程序（为常规输出添加前缀的代码部分），样式表，或用户选项。
--pages range	打印 range 范围内的页面。
--right-footer text	设置右页脚为 text。
--right-title text	设置页面右标题为 text。
--rows number	将所有页面排列成 number 排，默认为1。
-B	没有页眉。
-b text	设置页眉为 text。
-f size	使用字体大小为 size 号。
-l number	设置每行字符数为 number。此项和 -L 选项（见下方）可以给文件用其他程序来更准确

	的分页，如 pr。
-L number	设置每页行数为 number。
-M name	使用打印媒介的名称 — 例如，A4。
-n number	每页输出 number 份。
-o file	输出到文件 file。如果指定为 - ，则输出到标准输出。
-P printer	使用打印机 printer。如果未指定，则使用系统默认打印机。
-R	纵向打印。
-r	横向打印。
-T number	设置制表位为每 number 字符。
-u text	用 text 作为页面底图（水印）。

以上只是对 a2ps 的总结，更多的选项尚未列出。

注意：a2ps 目前仍在不断的开发中。就我的测试而言，不同版本之间都多少有所变化。CentOS 4 中输出总是默认为标准输出。在 CentOS 4 和 Fedora 10 中，尽管程序配置信纸为默认媒介，输出还是默认为 A4 纸。我可以明确的指定需要的选项来解决这些问题。Ubuntu 8.04 中，a2ps 表现的正如参考文档中所述。 另外，我们也要注意到另一个转换文本为 PostScript 的输出格式化工具，名叫 enscript。它具有许多相同的格式化和打印功能，但和 a2ps 唯一的不同在于，它只能处理纯文本的输入。

监视和控制打印任务

由于 Unix 打印系统的设计是能够处理多用户的多重打印任务，CUPS 也是如此设计的。每台打印机都有一个打印队列，其中的任务直到传送到打印机才停下并进行打印。CUPS 支持一些命令行程序来管理打印机状态和打印队列。想 lpr 和 lp 这样的管理程序都是以 Berkeley 和 System V 打印系统的相应程序为依据进行排列的。

lpstat——显示打印系统状态

lpstat 程序可用于确定系统中打印机的名字和有效性。例如，我们系统中有一台实体打印机（名叫 printer）和一台 PDF 虚拟打印机（名叫 PDF），我们可以像这样查看打印机状态：

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 05 Dec 2011 03:05:59 PM EST
printer accepting requests since Tue 21 Feb 2012 08:43:22 AM EST
```

接着，我们可以查看打印系统更具体的配置信息：

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

上例中，我们看到 printer 是系统默认的打印机，其本身是一台网络打印机，使用网络打印协议（ipp://）通过网络连接到名为 print-server 的系统。 lpstat 的常用选项列于表22-5。

表22－5：常用 lpstat 选项

选项	描述
-a [printer...]	显示 printer 打印机的队列。这里显示的状态是打印机队列承受任务的能力，而不是实体打印机的状态。若未指定打印机，则显示所有打印队列。
-d	显示系统默认打印机的名称。
-p [printer...]	显示 printer 指定的打印机的状态。若未指定打印机，则显示所有打印机状态。
-r	显示打印系统的状态。

-s	显示汇总状态。
-t	显示完整状态报告。

lpq——显示打印机队列状态

使用 lpq 程序可以查看打印机队列的状态，从中我们可以看到队列的状态和所包含的打印任务。下面的例子显示了一台名叫 printer 的系统默认打印机包含一个空队列的情况：

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

如果我们不指定打印机（用 -P 参数），就会显示系统默认打印机。如果给打印机添加一项任务再查看队列，我们就会看到下列结果：

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank     Owner   Job     File(s)          Total Size
active   me      603     (stdin)          1024 bytes
```

lprm 和 cancel——取消打印任务

CUPS 提供两个程序来从打印队列中终止并移除打印任务。一个是 Berkeley 风格的（lprm），另一个是 System V 的（cancel）。在支持的选项上两者有较小的区别但是功能却几乎相同。以上面的打印任务为例，我们可以像这样终止并移除任务：

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

每个命令都有选项可用于移除某用户、某打印机或多个任务号的所有任务，相应的参考手册中都有详细的介绍。

编译程序

在这一章中，我们将看一下如何通过编译源代码来创建程序。源代码的可用性是至关重要的自由，从而使得 Linux 成为可能。 整个 Linux 开发生态圈就是依赖于开发者之间的自由交流。对于许多桌面用户来说，编译是一种失传的艺术。以前很常见， 但现在，由系统发行版提供商维护巨大的预编译的二进制仓库，准备供用户下载和使用。在写这篇文章的时候，Debian 仓库（最大的发行版之一）包含了几乎23,000个预编译的包。

那么为什么要编译软件呢？ 有两个原因：

1. 可用性。尽管系统发行版仓库中已经包含了大量的预编译程序，但是一些发行版本不可能包含所有期望的应用。 在这种情况下，得到所期望程序的唯一方式是编译程序源码。

2. 及时性。虽然一些系统发行版专门打包前沿版本的应用程序，但是很多不是。这意味着， 为了拥有一个最新版本的程序，编译是必需的。

从源码编译软件可以变得非常复杂且具有技术性；许多用户难以企及。然而，许多编译任务是 相当简单的，只涉及到几个步骤。这都取决于程序包。我们将看一个非常简单的案例，为的是给大家提供一个对编译过程的整体认识， 并为那些愿意进一步学习的人们构筑一个起点。

我们将介绍一个新命令：

- make - 维护程序的工具

什么是编译？

简而言之，编译就是把源码（一个由程序员编写的人类可读的程序描述）翻译成计算机处理器的母语的过程。

计算机处理器（或 CPU）工作在一个非常基本的水平，执行用机器语言编写的程序。这是一种数值编码，描述非常小的操作， 比如“加这个字节”，“指向内存中的这个位置”，或者“复制这个字节”。

这些指令中的每一条都是用二进制表示的（1和0）。最早的计算机程序就是用这种数值编码写成的，这可能就 解释了为什么编写它们的程序员据说吸很多烟，喝大量咖啡，并带着厚厚的眼镜。这个问题克服了，随着汇编语言的出现， 汇编语言代替了数值编码（略微）简便地使用助记符，比如 CPY（复制）和 MOV（移动）。用汇编语言编写的程序通过 汇编器处理为机器语言。今天为了完成某些特定的程序任务，汇编语言仍在被使用，例如设备驱动和嵌入式系统。

下一步我们谈论一下什么是所谓的高级编程语言。之所以这样称呼它们，是因为它们可以让程序员少操心处理器的 一举一动，而更多关心如何解决手头的问题。早期的高级语言（二十世纪60年代期间研发的）包括 FORTRAN（为科学和技术问题而设计）和 COBOL（为商业应用而设计）。今天这两种语言仍在有限的使用。

虽然有许多流行的编程语言，两个占主导地位。大多数为现代系统编写的程序，要么用 C 编写，要么是用 C++ 编写。 在随后的例子中，我们将编写一个 C 程序。

用高级语言编写的程序，经过另一个称为编译器的程序的处理，会转换成机器语言。一些编译器把 高级指令翻译成汇编语言，然后使用一个汇编器完成翻译成机器语言的最后阶段。

一个称为链接的过程经常与编译结合在一起。有许多程序执行的常见任务。以打开文件为例。许多程序执行这个任务， 但是让每个程序实现它自己的打开文件功能，是很浪费资源的。更有意义的是，拥有单独的一段知道如何打开文件的程序， 并允许所有需要它的程序共享它。对常见任务提供支持由所谓的库完成。这些库包含多个程序，每个程序执行 一些可以由多个程序共享的常见任务。如果我们看一下 /lib 和 /usr/lib 目录，我们可以看到许多库定居在那里。 一个叫做链接器的程序用来在编译器的输出结果和要编译的程序所需的库之间建立连接。这个过程的最终结果是 一个可执行程序文件，准备使用。

所有的程序都是可编译的吗？

不是。正如我们所看到的，有些程序比如 shell 脚本就不需要编译。它们直接执行。 这些程序是用所谓的脚本或解释型语言编写的。近年来，这些语言变得越来越流行，包括 Perl， Python，PHP，Ruby，和许多其它语言。

脚本语言由一个叫做解释器的特殊程序执行。一个解释器输入程序文件，读取并执行程序中包含的每一条指令。 通常来说，

解释型程序执行起来要比编译程序慢很多。这是因为每次解释型程序执行时，程序中每一条源码指令都需要翻译， 而一个编译程序，一条源码指令只翻译一次，翻译后的指令会永久地记录到最终的执行文件中。

那么为什么解释型程序这样流行呢？对于许多编程任务来说，原因是“足够快”，但是真正的优势是一般来说开发解释型程序要比编译程序快速且容易。通常程序开发需要经历一个不断重复的写码，编译，测试周期。随着程序变得越来越大， 编译阶段会变得相当耗时。解释型语言删除了编译步骤，这样就加快了程序开发。

编译一个 C 语言

让我们编译一些东西。在我们行动之前，然而我们需要一些工具，像编译器，链接器，还有 make。 在 Linux 环境中，普遍使用的 C 编译器叫做 gcc（GNU C 编译器），最初由 Richard Stallman 写出来的。 大多数 Linux 系统发行版默认不安装 gcc。我们可以这样查看该编译器是否存在：

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

在这个例子中的输出结果表明安装了 gcc 编译器。

小提示： 你的系统发行版可能有一个用于软件开发的 meta-package（软件包的集合）。如果是这样的话， 考虑安装它，若你打算在你的系统中编译程序。若你的系统没有提供一个 meta-package，试着安装 gcc 和 make 工具包。 在许多发行版中，这就足够完成下面的练习了。

得到源码

为了我们的编译练习，我们将编译一个叫做 diction 的程序，来自 GNU 项目。这是一个小巧方便的程序， 检查文本文件的书写质量和样式。就程序而言，它相当小，且容易创建。

遵照惯例，首先我们要创建一个名为 src 的目录来存放我们的源码，然后使用 ftp 协议把源码下载下来。

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r-- 1 1003 65534 68940 Aug 28 1998 diction-0.7.tar.gz
-rw-r--r-- 1 1003 65534 90957 Mar 04 2002 diction-1.02.tar.gz
-rw-r--r-- 1 1003 65534 141062 Sep 17 2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz
(141062 bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

注意：因为我们是这个源码的“维护者”，当我们编译它的时候，我们把它保存在 ~/src 目录下。 由你的系统发行版源码会把源码安装在 /usr/src 目录下，而供多个用户使用的源码，通常安装在 /usr/local/src 目录下。

正如我们所看到的，通常提供的源码形式是一个压缩的 tar 文件。有时候称为 tarball，这个文件包含源码树，或者是组成源码的目录和文件的层次结构。当到达 ftp 站点之后，我们检查可用的 tar 文件列表，然后选择最新版本，下载。 使用 ftp 中的 get 命令，我们把文件从 ftp 服务器复制到本地机器。

一旦 tar 文件下载下来之后，必须打开。通过 tar 程序可以完成：

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11
diction-1.11.tar.gz
```

小提示：该 diction 程序，像所有的 GNU 项目软件，遵循着一定的源码打包标准。其它大多数在 Linux 生态系统中 可用的源码也遵循这个标准。该标准的一个条目是，当源码 tar 文件打开的时候，会创建一个目录，该目录包含了源码树， 并且这个目录将会命名为 project-x.xx，其包含了项目名称和它的版本号两项内容。这种方案能在系统中方便安装同一程序的多个版本。 然而，通常在打开 tarball 之前检验源码树的布局是个不错的主意。一些项目不会创建目录，反而，会把文件直接传递给当前目录。 这会把你的（除非组织良好的）src 目录弄得一片狼藉。为了避免这个，使用下面的命令，检查 tar 文件的内容：

```
tar tzvf tarfile | head
```

检查源码树

打开该 tar 文件，会创建一个新的目录，名为 diction-1.11。这个目录包含了源码树。让我们看一下里面的内容：

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess     diction.c          getopt.c      nl
config.h.in      diction.pot        getopt.h      nl.po
config.sub       diction.spec       getopt_int.h  README
configure        diction.spec.in    INSTALL       sentence.c
configure.in     diction.texi.in    install-sh    sentence.h
COPYING en       Makefile.in        style.1.in
de               en_GB              misc.c        style.c
de.po            en_GB.po           misc.h        test
diction.1.in     getopt1.c          NEWS
```

在源码树中，我们看到大量的文件。属于 GNU 项目的程序，还有其它许多程序都会，提供文档文件 README，INSTALL，NEWS，和 COPYING。

这些文件包含了程序描述，如何建立和安装它的信息，还有它许可条款。在试图建立程序之前，仔细阅读 README 和 INSTALL 文件，总是一个不错的主意。

在这个目录中，其它有趣的文件是那些以 .c 和 .h 为后缀的文件：

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c getopt1.c getopt.c misc.c sentence.c style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h getopt_int.h misc.h sentence.h
```

这些 .c 文件包含了由该软件包提供的两个 C 程序（style 和 diction），被分割成模块。这是一种常见做法，把大型程序 分解成更小，更容易管理的代码块。源码文件都是普通文本，可以用 less 命令查看：

```
[me@linuxbox diction-1.11]$ less diction.c
```

这些 .h 文件以头文件而著称。它们也是普通文件。头文件包含了程序的描述，这些程序被包括在源码文件或库中。 为了让编译器链接到模块，编译器必须接受所需的所有模块的描述，来完成整个程序。在 diction.c 文件的开头附近， 我们看到这行代码：

```
#include "getopt.h"
```

这行代码指示编译器去读取文件 getopt.h，因为它会读取 diction.c 中的源码，为的是“知道” getopt.c 中的内容。 getopt.c 文件提供由 style 和 diction 两个程序共享的代码。

在 getopt.h 的 include 语句上面，我们看到一些其它的 include 语句，比如这些：

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

这些也涉及到头文件，但是这些头文件居住在当前源码树的外面。它们由操作系统供给，来支持每个程序的编译。 如果我们看一下 /usr/include 目录，能看到它们：

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

当我们安装编译器的时候，这个目录中的头文件会被安装。

构建程序

大多数程序通过一个简单的，两个命令的序列建立：

```
./configure
make
```

这个 configure 程序是一个 shell 脚本，由源码树提供。它的工作是分析程序建立环境。大多数源码会设计为可移植的。 也就是说，它被设计成，能建立在多于一个的类 Unix 系统中。但是为了做到这一点，在建立程序期间，为了适应系统之间的差异， 源码可能需要经过轻微的调整。configure 也会检查是否安装了必要的外部工具和组件。让我们运行 configure 命令。 因为 configure 命令所在的位置不是位于 shell 通常期望程序所呆的地方，我们必须明确地告诉 shell 它的位置，通过 在命令之前加上 ./ 字符，来表明程序位于当前工作目录：

```
[me@linuxbox diction-1.11]$ ./configure
```

configure 将会输出许多信息，随着它测试和配置整个构建过程。当结束后，输出结果看起来像这样：

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

这里最重要的事情是没有错误信息。如果有错误信息，整个配置过程失败，然后程序不能构建直到修正了错误。

我们看到在我们的源码目录中 configure 命令创建了几个新文件。最重要一个是 Makefile。Makefile 是一个配置文件， 指示 make 程序究竟如何构建程序。没有它，make 程序就不能运行。Makefile 是一个普通文本文件，所以我们能查看它：

```
[me@linuxbox diction-1.11]$ less Makefile
```

这个 make 程序把一个 makefile 文件作为输入（通常命名为 Makefile），makefile 文件 描述了包括最终完成的程序的各组件之间的关系和依赖性。

makefile 文件的第一部分定义了变量，这些变量在该 makefile 后续章节中会被替换掉。例如我们看看这一行代码：

```
CC=                gcc
```

其定义了所用的 C 编译器是 gcc。文件后面部分，我们看到一个使用该变量的实例：

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

这里完成了一个替换操作，在程序运行时，$(CC) 的值会被替换成 gcc。大多数 makefile 文件由行组成，每行定义一个目标文件， 在这种情况下，目标文件是指可执行文件 diction，还有目标文件所依赖的文件。剩下的行描述了从目标文件的依赖组件中 创建目标文件所需的命令。在这个例子中，我们看到可执行文件 diction（最终的成品之一）依赖于文件 diction.o，sentence.o，misc.o，getopt.o，和 getopt1.o都存在。在 makefile 文件后面部分，我们看到 diction 文件所依赖的每一个文件做为目标文件的定义：

```
diction.o:       diction.c config.h getopt.h misc.h sentence.h
getopt.o:        getopt.c getopt.h getopt_int.h
getopt1.o:       getopt1.c getopt.h getopt_int.h
misc.o:          misc.c config.h misc.h
sentence.o:      sentence.c config.h misc.h sentence.h
style.o:         style.c config.h getopt.h misc.h sentence.h
```

然而，我们不会看到针对它们的任何命令。这个由一个通用目标解决，在文件的前面，描述了这个命令，用来把任意的 .c 文件编译成 .o 文件：

```
.c.o:
            $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

这些看起来非常复杂。为什么不简单地列出所有的步骤，编译完成每一部分？一会儿就知道答案了。同时， 让我们运行 make 命令并构建我们的程序：

```
[me@linuxbox diction-1.11]$ make
```

这个 make 程序将会运行，使用 Makefile 文件的内容来指导它的行为。它会产生很多信息。

当 make 程序运行结束后，现在我们将看到所有的目标文件出现在我们的目录中。

```
[me@linuxbox diction-1.11]$ ls
config.guess  de.po            en             en_GB          sentence.c
config.h      diction          en_GB.mo       en_GB.po       sentence.h
config.h.in   diction.1        getopt1.c      getopt1.o      sentence.o
config.log    diction.1.in     getopt.c       getopt.h       style
config.status diction.c        getopt_int.h   getopt.o       style.1
config.sub    diction.o        INSTALL        install-sh     style.1.in
configure     diction.pot      Makefile       Makefile.in    style.c
configure.in  diction.spec     misc.c         misc.h         style.o
COPYING       diction.spec.in  misc.o         NEWS           test
de            diction.texi     nl             nl.mo
de.mo         diction.texi.i   nl.po          README
```

在这些文件之中，我们看到 diction 和 style，我们开始要构建的程序。恭喜一切正常！我们刚才源码编译了 我们的第一个程序。但是出于好奇，让我们再运行一次 make 程序：

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

它只是产生这样一条奇怪的信息。怎么了？为什么它没有重新构建程序呢？啊，这就是 make 奇妙之处了。make 只是构建需要构建的部分，而不是简单地重新构建所有的内容。由于所有的目标文件都存在，make 确定没有任何事情需要做。 我们可以证明这一点，通过删除一个目标文件，然后再次运行 make 程序，看看它做些什么。让我们去掉一个中间目标文件：

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

我们看到 make 重新构建了 getopt.o 文件，并重新链接了 diction 和 style 程序，因为它们依赖于丢失的模块。 这种行为也指出了 make 程序的另一个重要特征：它保持目标文件是最新的。make 坚持目标文件要新于它们的依赖文件。 这个非常有意义，做为一名程序员，经常会更新一点儿源码，然后使用 make 来构建一个新版本的成品。make 确保 基于更新的代码构建了需要构建的内容。如果我们使用 touch 程序，来“更新”其中一个源码文件，我们看到发生了这样的事情：

```
[me@linuxboxdiction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2007-03-30 17:45 getopt.c
[me@linuxboxdiction-1.11]$ touch getopt.c
[me@linuxboxdiction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

运行 make 之后，我们看到目标文件已经更新于它的依赖文件：

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
```

make 程序这种智能地只构建所需要构建的内容的特性，对程序来说，是巨大的福利。虽然在我们的小项目中，节省的时间可能 不是非常明显，在庞大的工程中，它具有非常重大的意义。记住，Linux 内核（一个经历着不断修改和改进的程序）包含了几百万行代码。

安装程序

打包良好的源码经常包括一个特别的 make 目标文件，叫做 install。这个目标文件将在系统目录中安装最终的产品，以供使用。 通常，这个目录是 /usr/local/bin，为在本地所构建软件的传统安装位置。然而，通常普通用户不能写入该目录，所以我们必须变成超级用户， 来执行安装操作：

```
[me@linuxbox diction-1.11]$ sudo make install
After we perform the installation, we can check that the program is ready to go:
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
And there we have it!
```

总结

在这一章中，我们已经知道了三个简单命令：

```
./configure
make
make install
```

可以用来构建许多源码包。我们也知道了在程序维护过程中，make 程序起到了举足轻重的作用。make 程序可以用到 任何需要维护一个目标/依赖关系的任务中，不仅仅为了编译源代码。

拓展阅读

- Wikipedia 上面有关于编译器和 make 程序的好文章：

  http://en.wikipedia.org/wiki/Compiler

  http://en.wikipedia.org/wiki/Make_(software)

- GNU Make 手册

  http://www.gnu.org/software/make/manual/html_node/index.html

编写第一个 Shell 脚本

在前面的章节中，我们已经装备了一个命令行工具的武器库。虽然这些工具能够解决许多种计算问题， 但是我们仍然局限于在命令行中手动地一个一个使用它们。难道不是很棒，如果我们能够让 shell 来完成更多的工作？ 我们可以的。通过把我们的工具一起放置到我们自己设计的程序中，然后 shell 就会自己来执行这些复杂的任务序列。 通过编写 shell 脚本，我们让 shell 来做这些事情。

什么是 Shell 脚本？

最简单的解释，一个 shell 脚本就是一个包含一系列命令的文件。shell 读取这个文件，然后执行 文件中的所有命令，就好像这些命令已经直接被输入到了命令行中一样。

Shell 有些独特，因为它不仅是一个功能强大的命令行接口,也是一个脚本语言解释器。我们将会看到， 大多数能够在命令行中完成的任务也能够用脚本来实现，同样地，大多数能用脚本实现的操作也能够 在命令行中完成。

虽然我们已经介绍了许多 shell 功能，但只是集中于那些经常直接在命令行中使用的功能。 Shell 也提供了一些通常（但不总是）在编写程序时才使用的功能。

怎样编写一个 Shell 脚本

为了成功地创建和运行一个 shell 脚本，我们需要做三件事情：

1. 编写一个脚本。 Shell 脚本就是普通的文本文件。所以我们需要一个文本编辑器来书写它们。最好的文本 编辑器都会支持语法高亮，这样我们就能够看到一个脚本关键字的彩色编码视图。语法高亮会帮助我们查看某种常见 错误。为了编写脚本文件，vim，gedit，kate，和许多其它编辑器都是不错的候选者。

2. 使脚本文件可执行。 系统会相当挑剔不允许任何旧的文本文件被看作是一个程序，并且有充分的理由! 所以我们需要设置脚本文件的权限来允许其可执行。

3. 把脚本放置到 *shell* 能够找到的地方。 当没有指定可执行文件明确的路径名时，shell 会自动地搜索某些目录， 来查找此可执行文件。为了最大程度的方便，我们会把脚本放到这些目录当中。

脚本文件格式

为了保持编程传统，我们将创建一个 “hello world” 程序来说明一个极端简单的脚本。所以让我们启动 我们的文本编辑器，然后输入以下脚本：

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'
```

对于脚本中的最后一行，我们应该是相当的熟悉，仅仅是一个带有一个字符串参数的 echo 命令。 对于第二行也很熟悉。它看起来像一个注释，我们已经在许多我们检查和编辑过的配置文件中 看到过。关于 shell 脚本中的注释，它们也可以出现在文本行的末尾，像这样：

```
echo 'Hello World!' # This is a comment too
```

文本行中，# 符号之后的所有字符都会被忽略。

类似于许多命令，这也在命令行中起作用：

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

虽然很少在命令行中使用注释，但它们也能起作用。

我们脚本中的第一行文本有点儿神秘。它看起来它应该是一条注释，因为它起始于一个#符号，但是 它看起来太有意义，以至于不仅仅是注释。事实上，这个#!字符序列是一种特殊的结构叫做 shebang。 这个 shebang 被用来告诉操作系统将执行此脚本所用的解释器的名字。每个 shell 脚本都应该把这一文本行 作为它的第一行。

让我们把此脚本文件保存为 hello_world。

下一步我们要做的事情是让我们的脚本可执行。使用 chmod 命令，这很容易做到：

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1  me    me      63  2009-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1  me    me      63  2009-03-07 10:10 hello_world
```

对于脚本文件，有两个常见的权限设置；权限为755的脚本，则每个人都能执行，和权限为700的 脚本，只有文件所有者能够执行。注意为了能够执行脚本，脚本必须是可读的。

脚本文件位置

当设置了脚本权限之后，我们就能执行我们的脚本了：

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

为了能够运行此脚本，我们必须指定脚本文件明确的路径。如果我们没有那样做，我们会得到这样的提示：

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

为什么会这样呢？什么使我们的脚本不同于其它的程序？结果证明，什么也没有。我们的 脚本没有问题。是脚本存储位置的问题。回到第12章，我们讨论了 PATH 环境变量及其它在系统 查找可执行程序方面的作用。回顾一下，如果没有给出可执行程序的明确路径名，那么系统每次都会 搜索一系列的目录，来查找此可执行程序。这个 /bin 目录就是其中一个系统会自动搜索的目录。 这个目录列表被存储在一个名为 PATH 的环境变量中。这个 PATH 变量包含一个由冒号分隔开的目录列表。 我们可以查看 PATH 的内容：

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games
```

这里我们看到了我们的目录列表。如果我们的脚本驻扎在此列表中任意目录下，那么我们的问题将 会被解决。注意列表中的第一个目录，/home/me/bin。大多数的 Linux 发行版会配置 PATH 变量，让其包含 一个位于用户主目录下的 bin 目录，从而允许用户能够执行他们自己的程序。所以如果我们创建了 一个 bin 目录，并把我们的脚本放在这个目录下，那么这个脚本就应该像其它程序一样开始工作了：

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

它的确工作了。

如果这个 PATH 变量不包含这个目录，我们能够轻松地添加它，通过在我们的 .bashrc 文件中包含下面 这一行文本：

```
export PATH=~/bin:"$PATH"
```

当做了这个修改之后，它会在每个新的终端会话中生效。为了把这个修改应用到当前的终端会话中， 我们必须让 shell 重新读取这个.bashrc 文件。这可以通过 “sourcing”.bashrc 文件来完成：

```
[me@linuxbox ~]$ . .bashrc
```

这个点（.）命令是 source 命令的同义词，一个 shell 内部命令，用来读取一个指定的 shell 命令文件， 并把它看作是从键盘中输入的一样。

注意：在 Ubuntu 系统中，如果存在 ~/bin 目录，当执行用户的 .bashrc 文件时， Ubuntu 会自动地添加这个 ~/bin 目录到 PATH 变量中。所以在 Ubuntu 系统中，如果我们创建 了这个 ~/bin 目录，随后退出，然后再登录，一切会正常运行。

脚本文件的好去处

这个 ~/bin 目录是存放为个人所用脚本的好地方。如果我们编写了一个脚本，系统中的每个用户都可以使用它， 那么这个脚本的传统位置是 /usr/local/bin。系统管理员使用的脚本经常放到 /usr/local/sbin 目录下。 大多数情况下，本地支持的软件，不管是脚本还是编译过的程序，都应该放到 /usr/local 目录下， 而不是在 /bin 或 /usr/bin 目录下。这些目录都是由 Linux 文件系统层次结构标准指定，只包含由 Linux 发行商 所提供和维护的文件。

更多的格式技巧

严肃认真的脚本书写，一个关键目标是为了维护方便；也就是说，一个脚本可以轻松地被作者或其它 用户修改，使它适应变化的需求。使脚本容易阅读和理解是一种方便维护的方法。

长选项名称

我们学过的许多命令都以长短两种选项名称为特征。例如，这个 ls 命令有许多选项既可以用短形式也 可以用长形式来表示。例如：

```
[me@linuxbox ~]$ ls -ad
```

和：

```
[me@linuxbox ~]$ ls --all --directory
```

是等价的命令。为了减少输入，当在命令行中输入选项的时候，短选项更受欢迎，但是当书写脚本的时候， 长选项能提供可读性。

缩进和行继续符

当雇佣长命令的时候，通过把命令在几个文本行中展开，可以提高命令的可读性。 在第十八章中，我们看到了一个特别长的 find 命令实例：

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 ‘{}’ ‘;’ \) -or \( -type d -not -perm 0711 -exec chmod
0711 ‘{}’ ‘;’ \)
```

显然，这个命令有点儿难理解，当第一眼看到它的时候。在脚本中，这个命令可能会比较容易 理解，如果这样书写它：

```
find playground \
  \( \
    -type f \
    -not -perm 0600 \
    -exec chmod 0600 '{}' ';' \
  \) \
  -or \
  \( \
    -type d \
    -not -perm 0711 \
    -exec chmod 0711 '{}' ';' \
  \)
```

通过使用行继续符（反斜杠-回车符序列）和缩进，这个复杂命令的逻辑性更清楚地描述给读者。 这个技巧在命令行中同样生效，虽然很少使用它，因为输入和编辑这个命令非常麻烦。脚本和 命令行的一个区别是，脚本可能雇佣 tab 字符拉实现缩进，然而命令行却不能，因为 tab 字符被用来 激活自动补全功能。

> 为书写脚本配置 vim
>
> 这个 vim 文本编辑器有许多许多的配置设置。有几个常见的选项能够有助于脚本书写：
>
> *:syntax on*
>
> 打开语法高亮。通过这个设置，当查看脚本的时候，不同的 shell 语法元素会以不同的颜色 显示。这对于识别某些编程错误很有帮助。并且它看起来也很酷。注意为了这个功能起作用，你 必须安装了一个完整的 vim 版本，并且你编辑的文件必须有一个 shebang，来说明这个文件是 一个 shell 脚本。如果对于上面的命令，你遇到了困难，试试 *:set syntax=sh*。
>
> *:set hlsearch*
>
> 打开这个选项是为了高亮查找结果。比如说我们查找单词 “echo”。通过设置这个选项，这个 单词的每个实例会高亮显示。
>
> *:set tabstop=4*
>
> 设置一个 tab 字符所占据的列数。默认是8列。把这个值设置为4（一种常见做法）， 从而让长文本行更容易适应屏幕。
>
> *:set autoindent*
>
> 打开 "auto indent" 功能。这导致 vim 能对新的文本行缩进与刚输入的文本行相同的列数。 对于许多编程结构来说，这就加速了输入。停止缩进，输入 Ctrl-d。
>
> 通过把这些命令（没有开头的冒号字符）添加到你的 ~/.vimrc 文件中，这些改动会永久生效。

总结归纳

在这脚本编写的第一章中，我们已经看过怎样编写脚本，怎样让它们在我们的系统中轻松地执行。 我们也知道了怎样使用各种格式技巧来提高脚本的可读性（可维护性）。在以后的各章中，轻松维护 会作为编写好脚本的中心法则一次又一次地出现。

拓展阅读

- 查看各种各样编程语言的“Hello World”程序和实例：

  http://en.wikipedia.org/wiki/Hello_world

- 这篇 Wikipedia 文章讨论了更多关于 shebang 机制的内容：

  http://en.wikipedia.org/wiki/Shebang_(Unix)

启动一个项目

从这一章开始，我们将建设一个项目。这个项目的目的是为了了解怎样使用各种各样的 shell 功能来 创建程序，更重要的是，创建好程序。

我们将要编写的程序是一个报告生成器。它会显示系统的各种统计数据和它的状态，并将产生 HTML 格式的报告， 所以我们能通过网络浏览器，比如说 Firefox 或者 Konqueror，来查看这个报告。

通常，创建程序要经过一系列阶段，每个阶段会添加新的特性和功能。我们程序的第一个阶段将会 产生一个非常小的 HTML 网页，其不包含系统信息。随后我们会添加这些信息。

第一阶段：最小的文档

首先我们需要知道的事是一个规则的 HTML 文档的格式。它看起来像这样：

```
<HTML>
      <HEAD>
            <TITLE>Page Title</TITLE>
      </HEAD>
      <BODY>
            Page body.
      </BODY>
</HTML>
```

如果我们将这些内容输入到文本编辑器中，并把文件保存为 foo.html，然后我们就能在 Firefox 中 使用下面的 URL 来查看文件内容：

```
file:///home/username/foo.html
```

程序的第一个阶段将这个 HTML 文件输出到标准输出。我们可以编写一个程序，相当容易地完成这个任务。 启动我们的文本编辑器，然后创建一个名为 ~/bin/sys_info_page 的新文件：

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

随后输入下面的程序：

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>"
echo "      <HEAD>"
echo "            <TITLE>Page Title</TITLE>"
echo "      </HEAD>"
echo "      <BODY>"
echo "            Page body."
echo "      </BODY>"
echo "</HTML>"
```

我们第一次尝试解决这个问题，程序包含了一个 shebang，一条注释（总是一个好主意）和一系列的 echo 命令，每个命令负责输出一行文本。保存文件之后，我们将让它成为可执行文件，再尝试运行它：

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

当程序运行的时候，我们应该看到 HTML 文本在屏幕上显示出来，因为脚本中的 echo 命令会输出 发送到标准输出。我们再次运行这个程序，把程序的输出重定向到文件 sys_info_page.html 中， 从而我们可以通过网络浏览器来查看输出结果：

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

到目前为止，一切顺利。

在编写程序的时候，尽量做到简单明了，这总是一个好主意。当一个程序易于阅读和理解的时候， 维护它也就更容易，更不用说，通过减少键入量，可以使程序更容易书写了。我们当前的程序版本 工作正常，但是它可以更简单些。实际上，我们可以把所有的 echo 命令结合成一个 echo 命令，当然 这样能更容易地添加更多的文本行到程序的输出中。那么，把我们的程序修改为：

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>
    <HEAD>
          <TITLE>Page Title</TITLE>
    </HEAD>
    <BODY>
          Page body.
    </BODY>
</HTML>"
```

一个带引号的字符串可能包含换行符，因此可以包含多个文本行。Shell 会持续读取文本直到它遇到 右引号。它在命令行中也是这样工作的：

```
[me@linuxbox ~]$ echo "<HTML>
>         <HEAD>
              <TITLE>Page Title</TITLE>
>         </HEAD>
>         <BODY>
>               Page body.
>         </BODY>
></HTML>"
```

开头的 “>” 字符是包含在 PS2shell 变量中的 shell 提示符。每当我们在 shell 中键入多行语句的时候， 这个提示符就会出现。现在这个功能有点儿晦涩，但随后，当我们介绍多行编程语句时，它会派上大用场。

第二阶段：添加一点儿数据

现在我们的程序能生成一个最小的文档，让我们给报告添加些数据吧。为此，我们将做 以下修改：

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>
    <HEAD>
          <TITLE>System Information Report</TITLE>
    </HEAD>
    <BODY>
          <H1>System Information Report</H1>
    </BODY>
</HTML>"
```

我们增加了一个网页标题，并且在报告正文部分加了一个标题。

变量和常量

然而，我们的脚本存在一个问题。请注意字符串 “System Information Report” 是怎样被重复使用的？对于这个微小的脚本而言，它不是一个问题，但是让我们设想一下， 我们的脚本非常冗长，并且我们有许多这个字符串的实例。如果我们想要更换一个标题，我们必须 对脚本中的许多地方做修改，这会是很大的工作量。如果我们能整理一下脚本，让这个字符串只 出现一次而不是多次，会怎样呢？这样会使今后的脚本维护工作更加轻松。我们可以这样做：

```
#!/bin/bash
# Program to output a system information page
title="System Information Report"
echo "<HTML>
        <HEAD>
                <TITLE>$title</TITLE>
        </HEAD>
        <BODY>
                <H1>$title</H1>
        </BODY>
</HTML>"
```

通过创建一个名为 title 的变量，并把 “System Information Report”字符串赋值给它，我们就可以利用参数展开功能，把这个字符串放到文件中的多个位置。

那么，我们怎样来创建一个变量呢？很简单，我们只管使用它。当 shell 碰到一个变量的时候，它会 自动地创建它。这不同于许多编程语言，它们中的变量在使用之前，必须显式的声明或是定义。关于 这个问题，shell 要求非常宽松，这可能会导致一些问题。例如，考虑一下在命令行中发生的这种情形：

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool
[me@linuxbox ~]$
```

首先我们把“yes”赋给变量 foo，然后用 echo 命令来显示变量值。接下来，我们显示拼写错误的变量名 “fool”的变量值，然后得到一个空值。这是因为 shell 很高兴地创建了变量 fool，当 shell 遇到 fool 的时候， 并且赋给 fool 一个空的默认值。因此，我们必须小心谨慎地拼写！同样理解实例中究竟发生了什么事情也 很重要。从我们以前学习 shell 执行展开操作，我们知道这个命令：

```
[me@linuxbox ~]$ echo $foo
```

经历了参数展开操作，然后得到：

```
[me@linuxbox ~]$ echo yes
```

然而这个命令：

```
[me@linuxbox ~]$ echo $fool
```

展开为：

```
[me@linuxbox ~]$ echo
```

这个空变量展开值为空！对于需要参数的命令来说，这会引起混乱。下面是一个例子：

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after `foo.txt'
Try `cp --help' for more information.
```

我们给两个变量赋值，foo 和 foo1。然后我们执行 cp 操作，但是拼写错了第二个参数的名字。 参数展开之后，这个 cp 命令只接受到一个参数，虽然它需要两个。

有一些关于变量名的规则：

1. 变量名可由字母数字字符（字母和数字）和下划线字符组成。

2. 变量名的第一个字符必须是一个字母或一个下划线。

3. 变量名中不允许出现空格和标点符号。

单词 “variable” 意味着可变的值，并且在许多应用程序当中，都是以这种方式来使用变量的。然而， 我们应用程序中的变量，title，被用作一个常量。常量有一个名字且包含一个值，在这方面就 像是变量。不同之处是常量的值是不能改变的。在执行几何运算的应用程序中，我们可以把 PI 定义为 一个常量，并把 3.1415 赋值给它，用它来代替数字字面值。shell 不能辨别变量和常量；它们大多数情况下 是为了方便程序员。一个常用惯例是指定大写字母来表示常量，小写字母表示真正的变量。我们 将修改我们的脚本来遵从这个惯例：

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
echo "<HTML>
        <HEAD>
                <TITLE>$title</TITLE>
        </HEAD>
        <BODY>
                <H1>$title</H1>
        </BODY>
</HTML>"
```

我们亦借此机会，通过在标题中添加 shell 变量名 HOSTNAME，让标题变得活泼有趣些。 这个变量名是这台机器的网络名称。

注意：实际上，shell 确实提供了一种方法，通过使用带有 -r（只读）选项的内部命令 declare， 来强制常量的不变性。如果我们给 TITLE 这样赋值：

那么 shell 会阻止之后给 TITLE 的任意赋值。这个功能极少被使用，但为了很早之前的脚本， 它仍然存在。

给变量和常量赋值

这里是我们真正开始使用参数扩展知识的地方。正如我们所知道的，这样给变量赋值：

```
variable=value
```

这里的 variable 是变量的名字，value 是一个字符串。不同于一些其它的编程语言，shell 不会 在乎变量值的类型；它把它们都看作是字符串。通过使用带有 -i 选项的 declare 命令，你可以强制 shell 把 赋值限制为整型，但是，正如像设置变量为只读一样，极少这样做。

注意在赋值过程中，变量名，等号和变量值之间必须没有空格。那么，这些值由什么组成呢？ 可以展开成字符串的任意值：

```
a=z                     # Assign the string "z" to variable a.
b="a string"            # Embedded spaces must be within quotes.
c="a string and $b"     # Other expansions such as variables can be
                        # expanded into the assignment.

d=$(ls -l foo.txt)      # Results of a command.
e=$((5 * 7))            # Arithmetic expansion.
f="\t\ta string\n"      # Escape sequences such as tabs and newlines.
```

可以在同一行中对多个变量赋值：

```
a=5 b="a string"
```

在参数展开过程中，变量名可能被花括号 “{}” 包围着。由于变量名周围的上下文，其变得不明确的情况下，这会很有帮助。这里，我们试图把一个文件名从 myfile 改为 myfile1，使用一个变量：

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: missing destination file operand after `myfile'
Try `mv --help' for more information.
```

这种尝试失败了，因为 shell 把 mv 命令的第二个参数解释为一个新的（并且空的）变量。通过这种方法 可以解决这个问题：

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

通过添加花括号，shell 不再把末尾的1解释为变量名的一部分。

我们将利用这个机会来添加一些数据到我们的报告中，即创建包括的日期和时间，以及创建者的用户名：

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
        </BODY>
</HTML>"
```

我们已经知道了两种不同的文本输出方法，两种方法都使用了 echo 命令。还有第三种方法，叫做 here document 或者 here script。一个 here document 是另外一种 I/O 重定向形式，我们 在脚本文件中嵌入正文文本，然后把它发送给一个命令的标准输入。它这样工作：

```
command << token
text
token
```

这里的 command 是一个可以接受标准输入的命令名，token 是一个用来指示嵌入文本结束的字符串。 我们将修改我们的脚本，来使用一个 here document:

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
cat << _EOF_
<HTML>
        <HEAD>
              <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
              <H1>$TITLE</H1>
              <P>$TIME_STAMP</P>
        </BODY>
</HTML>
_EOF_
```

取代 echo 命令，现在我们的脚本使用 cat 命令和一个 here document。这个字符串 _EOF_（意思是“文件结尾”，一个常见用法）被选作为 token，并标志着嵌入文本的结尾。注意这个 token 必须在一行中单独出现，并且文本行中 没有末尾的空格。

那么使用一个 here document 的优点是什么呢？它很大程度上和 echo 一样，除了默认情况下，here documents 中的单引号和双引号会失去它们在 shell 中的特殊含义。这里有一个命令中的例子：

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

正如我们所见到的，shell 根本没有注意到引号。它把它们看作是普通的字符。这就允许我们 在一个 here document 中可以随意的嵌入引号。对于我们的报告程序来说，这将是非常方便的。

Here documents 可以和任意能接受标准输入的命令一块使用。在这个例子中，我们使用了 一个 here document 将一系列的命令传递到这个 ftp 程序中，为的是从一个远端 FTP 服务器中得到一个文件：

```
#!/bin/bash
# Script to retrieve a file via FTP
FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz
ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

如果我们把重定向操作符从 “<\<” 改为 “<\<-”，shell 会忽略在此 here document 中开头的 tab 字符。 这就能缩进一个 here document，从而提高脚本的可读性：

```
#!/bin/bash
# Script to retrieve a file via FTP
FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz
ftp -n <<- _EOF_
    open $FTP_SERVER
    user anonymous me@linuxbox
    cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
_EOF_
ls -l $REMOTE_FILE
```

总结归纳

在这一章中，我们启动了一个项目，其带领我们领略了创建一个成功脚本的整个过程。 同时我们介绍了变量和常量的概念，以及怎样使用它们。它们是我们将找到的众多参数展开应用程序中的第一批实例。 我们也知道了怎样从我们的脚本文件中产生输出，及其各种各样嵌入文本块的方法。

拓展阅读

- 关于 HTML 的更多信息，查看下面的文章和教材：

  http://en.wikipedia.org/wiki/Html

  http://en.wikibooks.org/wiki/HTML_Programming

  http://html.net/tutorials/html/

- Bash 手册包括一节“HERE DOCUMENTS”的内容，其详细的讲述了这个功能。

自顶向下设计

随着程序变得更加庞大和复杂，设计，编码和维护它们也变得更加困难。对于任意一个大项目而言， 把繁重，复杂的任务分割为细小且简单的任务，往往是一个好主意。想象一下，我们试图描述 一个平凡无奇的工作，一位火星人要去市场买食物。我们可能通过下面一系列步骤来形容整个过程：

- 上车
- 开车到市场
- 停车
- 买食物
- 回到车中
- 开车回家
- 回到家中

然而，火星人可能需要更详细的信息。我们可以进一步细化子任务"停车"为这些步骤：

- 找到停车位
- 开车到停车位
- 关闭引擎
- 拉紧手刹
- 下车
- 锁车

这个"关闭引擎"子任务可以进一步细化为这些步骤，包括"关闭点火装置"，"移开点火匙"等等，直到 已经完整定义了要去市场买食物整个过程的每一个步骤。

这种先确定上层步骤，然后再逐步细化这些步骤的过程被称为自顶向下设计。这种技巧允许我们 把庞大而复杂的任务分割为许多小而简单的任务。自顶向下设计是一种常见的程序设计方法， 尤其适合 shell 编程。

在这一章中，我们将使用自顶向下的设计方法来进一步开发我们的报告产生器脚本。

目前我们的脚本执行以下步骤来产生这个 HTML 文档：

- 打开网页
- 打开网页标头
- 设置网页标题
- 关闭网页标头
- 打开网页主体部分
- 输出网页标头
- 输出时间戳
- 关闭网页主体

- 关闭网页

为了下一阶段的开发，我们将在步骤7和8之间添加一些额外的任务。这些将包括：

- 系统正常运行时间和负载。这是自上次关机或重启之后系统的运行时间，以及在几个时间间隔内当前运行在处理 中的平均任务量。

- 磁盘空间。系统中存储设备的总使用量。

- 主目录空间。每个用户所使用的存储空间数量。

如果对于每一个任务，我们都有相应的命令，那么通过命令替换，我们就能很容易地把它们添加到我们的脚本中：

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </BODY>
</HTML>
_EOF_
```

我们能够用两种方法来创建这些额外的命令。我们可以分别编写三个脚本，并把它们放置到 环境变量 PATH 所列出的目录下，或者我们也可以把这些脚本作为 shell 函数嵌入到我们的程序中。 我们之前已经提到过，shell 函数是位于其它脚本中的“微脚本”，作为自主程序。Shell 函数有两种语法形式：

```
function name {
    commands
    return
}
and
name () {
    commands
    return
}
```

这里的 name 是函数名，commands 是一系列包含在函数中的命令。

两种形式是等价的，可以交替使用。下面我们将查看一个说明 shell 函数使用方法的脚本：

```
1     #!/bin/bash
2
3     # Shell function demo
4
5     function funct {
6         echo "Step 2"
7         return
8     }
9
10    # Main program starts here
11
12    echo "Step 1"
13    funct
14    echo "Step 3"
```

随着 shell 读取这个脚本，它会跳过第1行到第11行的代码，因为这些文本行由注释和函数定义组成。 从第12行代码开始执行，有一个 echo 命令。第13行会调用 shell 函数 funct，然后 shell 会执行这个函数， 就如执行其它命令一样。这样程序控制权会转移到第六行，执行第二个 echo 命令。然后再执行第7行。 这个 return 命令终止这个函数，并把控制权交给函数调用之后的代码（第14行），从而执行最后一个 echo 命令。注意为了使函数调用被识别出是 shell 函数，而不是被解释为外部程序的名字，所以在脚本中 shell 函数定义必须出现在函数调用之前。

我们将给脚本添加最小的 shell 函数定义：

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
report_uptime () {
    return
}
report_disk_space () {
    return
}
report_home_space () {
    return
}
cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </BODY>
</HTML>
_EOF_
```

Shell 函数的命名规则和变量一样。一个函数必须至少包含一条命令。这条 return 命令（是可选的）满足要求。

局部变量

目前我们所写的脚本中，所有的变量（包括常量）都是全局变量。全局变量在整个程序中保持存在。 对于许多事情来说，这很好，但是有时候它会使 shell 函数的使用变得复杂。在 shell 函数中，经常期望 会有局部变量。局部变量只能在定义它们的 shell 函数中使用， 并且一旦 shell 函数执行完毕，它们就不存在了。

拥有局部变量允许程序员使用的局部变量名，可以与已存在的变量名相同，这些变量可以是全局变量， 或者是其它 shell 函数中的局部变量，却不必担心潜在的名字冲突。

这里有一个实例脚本，其说明了怎样来定义和使用局部变量：

```
#!/bin/bash
# local-vars: script to demonstrate local variables
foo=0 # global variable foo
funct_1 () {
    local foo  # variable foo local to funct_1
    foo=1
    echo "funct_1: foo = $foo"
}
funct_2 () {
    local foo  # variable foo local to funct_2
    foo=2
    echo "funct_2: foo = $foo"
}
echo "global:  foo = $foo"
funct_1
echo "global: foo = $foo"
funct_2
echo "global: foo = $foo"
```

正如我们所看到的，通过在变量名之前加上单词 local，来定义局部变量。这就创建了一个只对其所在的 shell 函数起作用的变量。在这个 shell 函数之外，这个变量不再存在。当我们运行这个脚本的时候， 我们会看到这样的结果：

```
[me@linuxbox ~]$ local-vars
global:  foo = 0
funct_1: foo = 1
global:  foo = 0
funct_2: foo = 2
global:  foo = 0
```

我们看到对两个 shell 函数中的局部变量 foo 赋值，不会影响到在函数之外定义的变量 foo 的值。

这个功能就允许 shell 函数能保持各自以及与它们所在脚本之间的独立性。这个非常有价值，因为它帮忙 阻止了程序各部分之间的相互干涉。这样 shell 函数也可以移植。也就是说，按照需求， shell 函数可以在脚本之间进行剪切和粘贴。

保持脚本运行

当开发程序的时候，保持程序的可执行状态非常有用。这样做，并且经常测试，我们就可以在程序 开发过程的早期检测到错误。这将使调试问题容易多了。例如，如果我们运行这个程序，做一个小的修改， 然后再次执行这个程序，最后发现一个问题，非常有可能这个最新的修改就是问题的来源。通过添加空函数， 程序员称之为占位符，我们可以在早期阶段证明程序的逻辑流程。当构建一个占位符的时候， 能够包含一些为程序员提供反馈信息的代码是一个不错的主意，这些信息展示了正在执行的逻辑流程。 现在看一下我们脚本的输出结果：

```
[me@linuxbox ~]$ sys_info_page
<HTML>
<HEAD>
<TITLE>System Information Report For twin2</TITLE>
</HEAD>
<BODY>
<H1>System Information Report For linuxbox</H1>
<P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>

</BODY>
</HTML>
```

我们看到时间戳之后的输出结果中有一些空行，但是我们不能确定这些空行产生的原因。如果我们 修改这些函数，让它们包含一些反馈信息：

```
report_uptime () {
  echo "Function report_uptime executed."
  return
}
report_disk_space () {
  echo "Function report_disk_space executed."
  return
}
report_home_space () {
  echo "Function report_home_space executed."
  return
}
```

然后再次运行这个脚本：

```
[me@linuxbox ~]$ sys_info_page
<HTML>
<HEAD>
<TITLE>System Information Report For linuxbox</TITLE>
</HEAD>
<BODY>
<H1>System Information Report For linuxbox</H1>
<P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>
Function report_uptime executed.
Function report_disk_space executed.
Function report_home_space executed.
</BODY>
</HTML>
```

现在我们看到，事实上，执行了三个函数。

我们的函数框架已经各就各位并且能工作，是时候更新一些函数代码了。首先，是 report_uptime 函数：

```
report_uptime () {
  cat <<- _EOF_
  <H2>System Uptime</H2>
  <PRE>$(uptime)</PRE>
  _EOF_
  return
}
```

这些代码相当直截了当。我们使用一个 here 文档来输出标题和 uptime 命令的输出结果，命令结果被 <PRE> 标签包围， 为的是保持命令的输出格式。这个 report_disk_space 函数类似：

```
report_disk_space () {
  cat <<- _EOF_
  <H2>Disk Space Utilization</H2>
  <PRE>$(df -h)</PRE>
  _EOF_
  return
}
```

这个函数使用 df -h 命令来确定磁盘空间的数量。最后，我们将建造 report_home_space 函数：

```
report_home_space () {
  cat <<- _EOF_
  <H2>Home Space Utilization</H2>
  <PRE>$(du -sh /home/*)</PRE>
  _EOF_
  return
}
```

我们使用带有 -sh 选项的 du 命令来完成这个任务。然而，这并不是此问题的完整解决方案。虽然它会 在一些系统（例如 Ubuntu）中起作用，但是在其它系统中它不工作。这是因为许多系统会设置主目录的 权限，以此阻止其它用户读取它们，这是一个合理的安全措施。在这些系统中，这个 report_home_space 函数， 只有用超级用户权限执行我们的脚本时，才会工作。一个更好的解决方案是让脚本能根据用户的使用权限来 调整自己的行为。我们将在下一章中讨论这个问题。

> 你的 *.bashrc* 文件中的 *shell* 函数
>
> Shell 函数是更为完美的别名替代物，实际上是创建较小的个人所用命令的首选方法。别名 非常局限于命令的种类和它们支持的 shell 功能，然而 shell 函数允许任何可以编写脚本的东西。 例如，如果我们喜欢 为我们的脚本开发的这个 report_disk_space shell 函数，我们可以为我们的 .bashrc 文件 创建一个相似的名为 ds 的函数：

```
ds () {
  echo “Disk Space Utilization For $HOSTNAME”
  df -h
}
```

总结归纳

这一章中，我们介绍了一种常见的程序设计方法，叫做自顶向下设计，并且我们知道了怎样 使用 shell 函数按照要求来完成逐步细化的任务。我们也知道了怎样使用局部变量使 shell 函数 独立于其它函数，以及其所在程序的其它部分。这就有可能使 shell 函数以可移植的方式编写， 并且能够重复使用，通过把它们放置到多个程序中；节省了大量的时间。

拓展阅读

- Wikipedia 上面有许多关于软件设计原理的文章。这里是一些好文章：

  http://en.wikipedia.org/wiki/Top-down_design

  http://en.wikipedia.org/wiki/Subroutines

流程控制：if 分支结构

在上一章中，我们遇到一个问题。怎样使我们的报告生成器脚本能适应运行此脚本的用户的权限？这个问题的解决方案要求我们能找到一种方法，在脚本中基于测试条件结果，来“改变方向”。用编程术语表达，就是我们需要程序可以分支。让我们考虑一个简单的用伪码表示的逻辑实例，伪码是一种模拟的计算机语言，为的是便于人们理解：

```
X=5
If X = 5, then:
Say “X equals 5.”
Otherwise:
Say “X is not equal to 5.”
```

这就是一个分支的例子。根据条件，“Does X = 5?” 做一件事情，“Say X equals 5,” 否则，做另一件事情，“Say X is not equal to 5.”

使用 shell，我们可以编码上面的逻辑，如下所示：

```
x=5
if [ $x = 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

或者我们可以直接在命令行中输入以上代码（略有缩短）：

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does
not equal 5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does
not equal 5"; fi
does not equal 5
```

在这个例子中，我们执行了两次这个命令。第一次是，把 x 的值设置为5，从而导致输出字符串“equals 5”, 第二次是，把 x 的值设置为0，从而导致输出字符串“does not equal 5”。

这个 if 语句语法如下：

```
if commands; then
     commands
[elif commands; then
     commands...]
[else
     commands]
fi
```

这里的 commands 是指一系列命令。第一眼看到会有点儿困惑。但是在我们弄清楚这些语句之前，我们 必须看一下 shell 是如何评判一个命令的成功与失败的。

退出状态

当命令执行完毕后，命令（包括我们编写的脚本和 shell 函数）会给系统发送一个值，叫做退出状态。 这个值是一个 0 到 255 之间的整数，说明命令执行成功或是失败。按照惯例，一个零值说明成功，其它所有值说明失败。 Shell 提供了一个参数，我们可以用它检查退出状态。用具体实例看一下：

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

在这个例子中，我们执行了两次 ls 命令。第一次，命令执行成功。如果我们显示参数 `$?` 的值，我们 看到它是零。我们第二次执行 ls 命令的时候，产生了一个错误，并再次查看参数 `$?`。这次它包含一个 数字 2，表明这个命令遇到了一个错误。有些命令使用不同的退出值，来诊断错误，而许多命令当 它们执行失败的时候，会简单地退出并发送一个数字1。手册页中经常会包含一章标题为"退出状态"的内容， 描述了使用的代码。然而，一个零总是表明成功。

这个 shell 提供了两个极其简单的内部命令，它们不做任何事情，除了以一个零或1退出状态来终止执行。 True 命令总是执行成功，而 false 命令总是执行失败：

```
[me@linuxbox~]$ true
[me@linuxbox~]$ echo $?
0
[me@linuxbox~]$ false
[me@linuxbox~]$ echo $?
1
```

我们能够使用这些命令，来看一下 if 语句是怎样工作的。If 语句真正做的事情是计算命令执行成功或失败：

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

当 if 之后的命令执行成功的时候，命令 echo "It's true." 将会执行，否则此命令不执行。 如果 if 之后跟随一系列命令，则将计算列表中的最后一个命令：

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
3
```

测试

到目前为止，经常与 if 一块使用的命令是 test。这个 test 命令执行各种各样的检查与比较。 它有两种等价模式：

```
test expression
```

比较流行的格式是：

```
[ expression ]
```

这里的 expression 是一个表达式，其执行结果是 true 或者是 false。当表达式为真时，这个 test 命令返回一个零 退出状态，当表达式为假时，test 命令退出状态为1。

文件表达式

以下表达式被用来计算文件状态：

表28-1: 测试文件表达式

表达式	如果为真
file1 -ef file2	file1 和 file2 拥有相同的索引号（通过硬链接两个文件名指向相同的文件）。
file1 -nt file2	file1新于 file2。
file1 -ot file2	file1早于 file2。
-b file	file 存在并且是一个块（设备）文件。
-c file	file 存在并且是一个字符（设备）文件。
-d file	file 存在并且是一个目录。
-e file	file 存在。
-f file	file 存在并且是一个普通文件。
-g file	file 存在并且设置了组 ID。
-G file	file 存在并且由有效组 ID 拥有。
-k file	file 存在并且设置了它的“sticky bit”。
-L file	file 存在并且是一个符号链接。
-O file	file 存在并且由有效用户 ID 拥有。
-p file	file 存在并且是一个命名管道。
-r file	file 存在并且可读（有效用户有可读权限）。
-s file	file 存在且其长度大于零。
-S file	file 存在且是一个网络 socket。
-t fd	fd 是一个定向到终端／从终端定向的文件描述符 。 这可以被用来决定是否重定向了标准输入／输出错误。
-u file	file 存在并且设置了 setuid 位。
-w file	file 存在并且可写（有效用户拥有可写权限）。
-x file	file 存在并且可执行（有效用户有执行／搜索权限）。

这里我们有一个脚本说明了一些文件表达式：

```
#!/bin/bash
# test-file: Evaluate the status of a file
FILE=~/.bashrc
if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi
exit
```

这个脚本会计算赋值给常量 FILE 的文件，并显示计算结果。对于此脚本有两点需要注意。第一个， 在表达式中参数 `$FILE`

是怎样被引用的。引号并不是必需的，但这是为了防范空参数。如果 `$FILE` 的参数展开 是一个空值，就会导致一个错误（操作符将会被解释为非空的字符串而不是操作符）。用引号把参数引起来就 确保了操作符之后总是跟随着一个字符串，即使字符串为空。第二个，注意脚本末尾的 exit 命令。 这个 exit 命令接受一个单独的，可选的参数，其成为脚本的退出状态。当不传递参数时，退出状态默认为零。 以这种方式使用 exit 命令，则允许此脚本提示失败如果 `$FILE` 展开成一个不存在的文件名。这个 exit 命令 出现在脚本中的最后一行，是一个当一个脚本“运行到最后”（到达文件末尾），不管怎样， 默认情况下它以退出状态零终止。

类似地，通过带有一个整数参数的 return 命令，shell 函数可以返回一个退出状态。如果我们打算把 上面的脚本转变为一个 shell 函数，为了在更大的程序中包含此函数，我们用 return 语句来代替 exit 命令， 则得到期望的行为：

```
test_file () {
    # test-file: Evaluate the status of a file
    FILE=~/.bashrc
    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
            echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
            echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi
}
```

字符串表达式

以下表达式用来计算字符串：

表28-2: 测试字符串表达式

表达式	如果为真...
string	string 不为 null。
-n string	字符串 string 的长度大于零。
-z string	字符串 string 的长度为零。
string1 = string2 string1 == string2	string1 和 string2 相同. 单或双等号都可以，不过双等号更受欢迎。
string1 != string2	string1 和 string2 不相同。
string1 > string2	sting1 排列在 string2 之后。
string1 < string2	string1 排列在 string2 之前。

警告：这个 < 和 > 表达式操作符必须用引号引起来（或者是用反斜杠转义）， 当与 test 一块使用的时候。如果不这样，它们会被 shell 解释为重定向操作符，造成潜在地破坏结果。 同时也要注意虽然 bash 文档声明排序遵从当前语系的排列规则，但并不这样。将来的 bash 版本，包含 4.0， 使用 ASCII（POSIX）排序规则。

这是一个演示这些问题的脚本：

```
#!/bin/bash
# test-string: evaluate the value of a string
ANSWER=maybe
if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi
if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

在这个脚本中，我们计算常量 ANSWER。我们首先确定是否此字符串为空。如果为空，我们就终止 脚本，并把退出状态设为零。注意这个应用于 echo 命令的重定向操作。其把错误信息 “There is no answer.” 重定向到标准错误，这是处理错误信息的“合理”方法。如果字符串不为空，我们就计算 字符串的值，看看它是否等于 “yes,” “no,” 或者 “maybe”。为此使用了 elif，它是 “else if” 的简写。 通过使用 elif，我们能够构建更复杂的逻辑测试。

整型表达式

下面的表达式用于整数：

表28-3: 测试整数表达式

表达式	如果为真...
integer1 -eq integer2	integer1 等于 integer2.
integer1 -ne integer2	integer1 不等于 integer2.
integer1 -le integer2	integer1 小于或等于 integer2.
integer1 -lt integer2	integer1 小于 integer2.
integer1 -ge integer2	integer1 大于或等于 integer2.
integer1 -gt integer2	integer1 大于 integer2.

这里是一个演示以上表达式用法的脚本：

```
#!/bin/bash
# test-integer: evaluate the value of an integer.
INT=-5
if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi
if [ $INT -eq 0 ]; then
    echo "INT is zero."
else
    if [ $INT -lt 0 ]; then
        echo "INT is negative."
    else
        echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
        echo "INT is even."
    else
        echo "INT is odd."
    fi
fi
```

这个脚本中有趣的地方是怎样来确定一个整数是偶数还是奇数。通过用模数2对数字执行求模操作， 就是用数字来除以2，并返回余数，从而知道数字是偶数还是奇数。

更现代的测试版本

目前的 bash 版本包括一个复合命令，作为加强的 test 命令替代物。它使用以下语法：

```
[[ expression ]]
```

这里，类似于 test，expression 是一个表达式，其计算结果为真或假。这个 `[[ ]]` 命令非常 相似于 test 命令（它支持所有的表达式），但是增加了一个重要的新的字符串表达式：

```
string1 =~ regex
```

其返回值为真，如果 string1 匹配扩展的正则表达式 regex。这就为执行比如数据验证等任务提供了许多可能性。 在我们前面的整数表达式示例中，如果常量 INT 包含除了整数之外的任何数据，脚本就会运行失败。这个脚本 需要一种方法来证明此常量包含一个整数。使用 `[[ ]]` 和 `=~` 字符串表达式操作符，我们能够这样来改进脚本：

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

通过应用正则表达式，我们能够限制 INT 的值只是字符串，其开始于一个可选的减号，随后是一个或多个数字。 这个表达式也消除了空值的可能性。

`[[ ]]` 添加的另一个功能是 `==` 操作符支持类型匹配，正如路径名展开所做的那样。例如：

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

这就使 `[[ ]]` 有助于计算文件和路径名。

(( )) - 为整数设计

除了 `[[ ]]` 复合命令之外，bash 也提供了 `(( ))` 复合命名，其有利于操作整数。它支持一套 完整的算术计算，我们将在第35章中讨论这个主题。

`(( ))` 被用来执行算术真测试。如果算术计算的结果是非零值，则一个算术真测试值为真。

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

使用`(( ))`，我们能够略微简化 test-integer2脚本，像这样：

```
#!/bin/bash
# test-integer2a: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if ((INT == 0)); then
        echo "INT is zero."
    else
        if ((INT < 0)); then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if (( ((INT % 2)) == 0)); then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

注意我们使用小于和大于符号，以及==用来测试是否相等。这是使用整数较为自然的语法了。也要 注意，因为复合命令`(( ))`是 shell 语法的一部分，而不是一个普通的命令，而且它只处理整数， 所以它能够通过名字识别出变量，而不需要执行展开操作。我们将在第35中进一步讨论`(( ))`命令 和相关的算术展开操作。

结合表达式

也有可能把表达式结合起来创建更复杂的计算。通过使用逻辑操作符来结合表达式。我们 在第18章中已经知道了这些，当我们学习 find 命令的时候。它们是用于 test 和`[[ ]]`三个逻辑操作。 它们是 AND，OR，和 NOT。test 和`[[ ]]`使用不同的操作符来表示这些操作：

表28-4: 逻辑操作符

操作符	测试	**[[ ]] and (( ))**
AND	-a	&&
OR	-o	||
NOT	!	!

这里有一个 AND 操作的示例。下面的脚本决定了一个整数是否属于某个范围内的值：

```
#!/bin/bash
# test-integer3: determine if an integer is within a
# specified range of values.
MIN_VAL=1
MAX_VAL=100
INT=50
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
        echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

我们也可以对表达式使用圆括号，为的是分组。如果不使用括号，那么否定只应用于第一个 表达式，而不是两个组合的表达式。用 test 可以这样来编码：

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

因为 test 使用的所有的表达式和操作符都被 shell 看作是命令参数（不像 `[[ ]]` 和 `(( ))` ）， 对于 bash 有特殊含义的字符，比如说 <，>，(，和)，必须引起来或者是转义。

知道了 test 和 `[[ ]]` 基本上完成相同的事情，哪一个更好呢？test 更传统（是 POSIX 的一部分）， 然而 `[[ ]]` 特定于 bash。知道怎样使用 test 很重要，因为它被非常广泛地应用，但是显然 `[[ ]]` 更 有助于，并更易于编码。

> 可移植性是头脑狭隘人士的心魔
>
> 如果你和“真正的” Unix 用户交谈，你很快就会发现他们大多数人不是非常喜欢 Linux。他们 认为 Linux 肮脏且不干净。Unix 追随者的一个宗旨是，一切都应“可移植的”。这意味着你编写 的任意一个脚本都应当无需修改，就能运行在任何一个类似于 Unix 的系统中。
>
> Unix 用户有充分的理由相信这一点。在 POSIX 之前，Unix 用户已经看到了命令的专有扩展以及 shell 对 Unix 世界的所做所为，他们自然会警惕 Linux 对他们心爱系统的影响。
>
> 但是可移植性有一个严重的缺点。它防碍了进步。它要求做事情要遵循“最低常见标准”。 在 shell 编程这种情况下，它意味着一切要与 sh 兼容，最初的 Bourne shell。
>
> 这个缺点是一个借口，专有软件供应商用它来证明他们的专利扩展，只有他们称他们为“创新”。 但是他们只是为他们的客户锁定设备。
>
> GNU 工具，比如说 bash，就没有这些限制。他们通过支持标准和普遍地可用性来鼓励可移植性。你几乎可以 在所有类型的系统中安装 bash 和其它的 GNU 工具，甚至是 Windows，而没有损失。所以就 感觉可以自由的使用 bash 的所有功能。它是真正的可移植。

控制操作符：分支的另一种方法

bash 支持两种可以执行分支任务的控制操作符。这个 `&&（AND）` 和 `||（OR）` 操作符作用如同 复合命令 `[[ ]]` 中的逻辑操作符。这是语法：

```
command1 && command2
```

和

```
command1 || command2
```

理解这些操作很重要。对于 && 操作符，先执行 command1，如果并且只有如果 command1 执行成功后， 才会执行 command2。对于 || 操作符，先执行 command1，如果并且只有如果 command1 执行失败后， 才会执行 command2。

在实际中，它意味着我们可以做这样的事情：

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

这会创建一个名为 temp 的目录，并且若它执行成功后，当前目录会更改为 temp。第二个命令会尝试 执行只有当 mkdir 命令执行成功之后。同样地，一个像这样的命令：

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

会测试目录 temp 是否存在，并且只有测试失败之后，才会创建这个目录。这种构造类型非常有助于在 脚本中处理错误，这个主题我们将会在随后的章节中讨论更多。例如，我们在脚本中可以这样做：

```
[ -d temp ] || exit 1
```

如果这个脚本要求目录 temp，且目录不存在，然后脚本会终止，并返回退出状态1。

总结

这一章开始于一个问题。我们怎样使 `sys_info_page` 脚本来检测是否用户拥有权限来读取所有的 主目录？根据我们的 if 知识，我们可以解决这个问题，通过把这些代码添加到 `report_home_space` 函数中：

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
        <H2>Home Space Utilization (All Users)</H2>
        <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
        <H2>Home Space Utilization ($USER)</H2>
        <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

我们计算 id 命令的输出结果。通过带有-u 选项的 id 命令，输出有效用户的数字用户 ID 号。 超级用户总是零，其它每个用户是一个大于零的数字。知道了这点，我们能够构建两种不同的 here 文档， 一个利用超级用户权限，另一个限制于用户拥有的主目录。

我们将暂别 `sys_info_page` 程序，但不要着急。它还会回来。同时，当我们继续工作的时候， 将会讨论一些我们需要的话题。

拓展阅读

bash 手册页中有几部分对本章中涵盖的主题提供了更详细的内容：

- Lists (讨论控制操作符 `||` 和 `&&`)
- Compound Commands (讨论 `[[ ]]` , `(( ))` 和 if)
- SHELL BUILTIN COMMANDS (讨论 test)

进一步，Wikipedia 中有一篇关于伪代码概念的好文章：

http://en.wikipedia.org/wiki/Pseudocode

读取键盘输入

到目前为止我们编写的脚本都缺乏一项在大多数计算机程序中都很常见的功能－交互性。也就是， 程序与用户进行交互的能力。虽然许多程序不必是可交互的，但一些程序却得到益处，能够直接 接受用户的输入。以这个前面章节中的脚本为例：

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
        echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

每次我们想要改变 INT 数值的时候，我们必须编辑这个脚本。如果脚本能请求用户输入数值，那 么它会更加有用处。在这个脚本中，我们将看一下我们怎样给程序增加交互性功能。

read – 从标准输入读取数值

这个 read 内部命令被用来从标准输入读取单行数据。这个命令可以用来读取键盘输入，当使用 重定向的时候，读取文件中的一行数据。这个命令有以下语法形式：

```
read [-options] [variable...]
```

这里的 options 是下面列出的可用选项中的一个或多个，且 variable 是用来存储输入数值的一个或多个变量名。 如果没有提供变量名，shell 变量 `REPLY` 会包含数据行。

基本上，read 会把来自标准输入的字段赋值给具体的变量。如果我们修改我们的整数求值脚本，让其使用 read，它可能看起来像这样：

```
#!/bin/bash
# read-integer: evaluate the value of an integer.
echo -n "Please enter an integer -> "
read int
if [[ "$int" =~ ^-?[0-9]+$ ]]; then
    if [ $int -eq 0 ]; then
        echo "$int is zero."
    else
        if [ $int -lt 0 ]; then
            echo "$int is negative."
        else
            echo "$int is positive."
        fi
        if [ $((int % 2)) -eq 0 ]; then
            echo "$int is even."
        else
            echo "$int is odd."
        fi
    fi
else
    echo "Input value is not an integer." >&2
    exit 1
fi
```

我们使用带有 -n 选项（其会删除输出结果末尾的换行符）的 echo 命令，来显示提示信息， 然后使用 read 来读入变量 int 的数值。运行这个脚本得到以下输出：

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

`read`可以给多个变量赋值，正如下面脚本中所示：

```
#!/bin/bash
# read-multiple: read multiple values from keyboard
echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5
echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

在这个脚本中，我们给五个变量赋值并显示其结果。注意当给定不同个数的数值后， `read` 怎样操作：

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

如果 `read` 命令接受到变量值数目少于期望的数字，那么额外的变量值为空，而多余的输入数据则会 被包含到最后一个变量中。如果 `read` 命令之后没有列出变量名，则一个 shell 变量， `REPLY`，将会包含 所有的输入：

```
#!/bin/bash
# read-single: read multiple values into default variable
echo -n "Enter one or more values > "
read
echo "REPLY = '$REPLY'"
```

这个脚本的输出结果是：

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

选项

read 支持以下选送：

表29-1: read 选项

选项	说明
-a array	把输入赋值到数组 array 中，从索引号零开始。我们 将在第36章中讨论数组问题。
-d delimiter	用字符串 delimiter 中的第一个字符指示输入结束，而不是一个换行符。
-e	使用 Readline 来处理输入。这使得与命令行相同的方式编辑输入。
-n num	读取 num 个输入字符，而不是整行。
-p prompt	为输入显示提示信息，使用字符串 prompt。
-r	Raw mode. 不把反斜杠字符解释为转义字符。
-s	Silent mode. 不会在屏幕上显示输入的字符。当输入密码和其它确认信息的时候，这会很有帮助。
-t seconds	超时. 几秒钟后终止输入。read 会返回一个非零退出状态，若输入超时。
-u fd	使用文件描述符 fd 中的输入，而不是标准输入。

使用各种各样的选项，我们能用 read 完成有趣的事情。例如，通过 -p 选项，我们能够提供提示信息：

```
#!/bin/bash
# read-single: read multiple values into default variable
read -p "Enter one or more values > "
echo "REPLY = '$REPLY'"
```

通过 -t 和 -s 选项，我们可以编写一个这样的脚本，读取“秘密”输入，并且如果在特定的时间内 输入没有完成，就终止输入。

```
#!/bin/bash
# read-secret: input a secret pass phrase
if read -t 10 -sp "Enter secret pass phrase > " secret_pass; then
    echo -e "\nSecret pass phrase = '$secret_pass'"
else
    echo -e "\nInput timed out" >&2
    exit 1
if
```

这个脚本提示用户输入一个密码，并等待输入10秒钟。如果在特定的时间内没有完成输入， 则脚本会退出并返回一个错误。因为包含了一个 -s 选项，所以输入的密码不会出现在屏幕上。

通常，shell 对提供给 read 的输入按照单词进行分离。正如我们所见到的，这意味着多个由一个或几个空格 分离开的单词在

输入行中变成独立的个体，并被 read 赋值给单独的变量。这种行为由 shell 变量 *IFS* （内部字符分隔符）配置。*IFS* 的默认值包含一个空格，一个 tab，和一个换行符，每一个都会把 字段分割开。

我们可以调整 *IFS* 的值来控制输入字段的分离。例如，这个 /etc/passwd 文件包含的数据行 使用冒号作为字段分隔符。通过把 *IFS* 的值更改为单个冒号，我们可以使 read 读取 /etc/passwd 中的内容， 并成功地把字段分给不同的变量。这个就是做这样的事情：

```
#!/bin/bash
# read-ifs: read fields from a file
FILE=/etc/passwd
read -p "Enter a user name > " user_name
file_info=$(grep "^$user_name:" $FILE)
if [ -n "$file_info" ]; then
    IFS=":" read user pw uid gid name home shell <<< "$file_info"
    echo "User = '$user'"
    echo "UID = '$uid'"
    echo "GID = '$gid'"
    echo "Full Name = '$name'"
    echo "Home Dir. = '$home'"
    echo "Shell = '$shell'"
else
    echo "No such user '$user_name'" >&2
    exit 1
fi
```

这个脚本提示用户输入系统中一个帐户的用户名，然后显示在文件 /etc/passwd/ 文件中关于用户记录的 不同字段。这个脚本包含两个有趣的文本行。 第一个是：

```
file_info=$(grep "^$user_name:" $FILE)
```

这一行把 grep 命令的输入结果赋值给变量 file_info。grep 命令使用的正则表达式 确保用户名只会在 /etc/passwd 文件中匹配一个文本行。

第二个有意思的文本行是：

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

这一行由三部分组成：一个变量赋值，一个带有一串参数的 read 命令，和一个奇怪的新的重定向操作符。 我们首先看一下变量赋值。

Shell 允许在一个命令之前立即发生一个或多个变量赋值。这些赋值为跟随着的命令更改环境变量。 这个赋值的影响是暂时的；只是在命令存在期间改变环境变量。在这种情况下，IFS 的值改为一个冒号。 另外，我们也可以这样编码：

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

我们先存储 IFS 的值，然后赋给一个新值，再执行 read 命令，最后把 IFS 恢复原值。显然，完成相同的任务， 在命令之前放置变量名赋值是一种更简明的方式。

这个 `<<<` 操作符指示一个 here 字符串。一个 here 字符串就像一个 here 文档，只是比较简短，由 单个字符串组成。在这个例子中，来自 /etc/passwd 文件的数据发送给 read 命令的标准输入。 我们可能想知道为什么选择这种相当晦涩的方法而不是：

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

> 你不能管道 *read*
>
> echo "foo" | read
>
> 虽然通常 `read` 命令接受标准输入，但是你不能这样做：
>
> 我们期望这个命令能生效，但是它不能。这个命令将显示成功，但是 `REPLY` 变量 总是为空。为什么会这样？
>
> 答案与 shell 处理管道线的方式有关系。在 bash（和其它 shells，例如 sh）中，管道线 会创建子 shell。它们是 shell 的副本，且用来执行命令的环境变量在管道线中。 上面示例中，read 命令将在子 shell 中执行。
>
> 在类似于 Unix 的系统中，子 shell 执行的时候，会为进程创建父环境的副本。当进程结束 之后，环境副本就会被破坏掉。这意味着一个子 shell 永远不能改变父进程的环境。read 赋值变量， 然后会变为环境的一部分。在上面的例子中，read 在它的子 shell 环境中，把 foo 赋值给变量 REPLY， 但是当命令退出后，子 shell 和它的环境将被破坏掉，这样赋值的影响就会消失。
>
> 使用 here 字符串是解决此问题的一种方法。另一种方法将在37章中讨论。

校正输入

从键盘输入这种新技能，带来了额外的编程挑战，校正输入。很多时候，一个良好编写的程序与 一个拙劣程序之间的区别就是程序处理意外的能力。通常，意外会以错误输入的形式出现。在前面 章节中的计算程序，我们已经这样做了一点儿，我们检查整数值，甄别空值和非数字字符。每次 程序接受输入的时候，执行这类的程序检查非常重要，为的是避免无效数据。对于 由多个用户共享的程序，这个尤为重要。如果一个程序只使用一次且只被作者用来执行一些特殊任务， 那么为了经济利益而忽略这些保护措施，可能会被原谅。即使这样，如果程序执行危险任务，比如说 删除文件，所以最好包含数据校正，以防万一。

这里我们有一个校正各种输入的示例程序：

```
#!/bin/bash
# read-validate: validate input
invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}
read -p "Enter a single item > "
# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input
# input is multiple items (invalid)
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input
# is input a valid filename?
if [[ $REPLY =~ ^[-[:alnum:]\._]+$ ]]; then
    echo "'$REPLY' is a valid filename."
    if [[ -e $REPLY ]]; then
        echo "And file '$REPLY' exists."
    else
        echo "However, file '$REPLY' does not exist."
    fi
    # is input a floating point number?
    if [[ $REPLY =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
        echo "'$REPLY' is a floating point number."
    else
        echo "'$REPLY' is not a floating point number."
    fi
    # is input an integer?
    if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then
        echo "'$REPLY' is an integer."
    else
        echo "'$REPLY' is not an integer."
    fi
else
    echo "The string '$REPLY' is not a valid filename."
fi
```

这个脚本提示用户输入一个数字。随后，分析这个数字来决定它的内容。正如我们所看到的，这个脚本 使用了许多我们已经讨论过的概念，包括 shell 函数，`[[ ]]`，`(( ))`，控制操作符 `&&`，以及 `if` 和 一些正则表达式。

菜单

一种常见的交互类型称为菜单驱动。在菜单驱动程序中，呈现给用户一系列选择，并要求用户选择一项。 例如，我们可以想象一个展示以下信息的程序：

```
Please Select:
1.Display System Information
2.Display Disk Space
3.Display Home Space Utilization
0.Quit
Enter selection [0-3] >
```

使用我们从编写 sys_info_page 程序中所学到的知识，我们能够构建一个菜单驱动程序来执行 上述菜单中的任务：

```
#!/bin/bash
# read-menu: a menu driven system information program
clear
echo "
Please Select:

    1. Display System Information
    2. Display Disk Space
    3. Display Home Space Utilization
    0. Quit
"
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi
```

The presence of multiple ` exit ` points in a program is generally a bad idea (it makes

从逻辑上讲，这个脚本被分为两部分。第一部分显示菜单和用户输入。第二部分确认用户反馈，并执行 选择的行动。注意脚本中使用的 exit 命令。在这里，在一个行动执行之后，exit 被用来阻止脚本执行不必要的代码。 通常在程序中出现多个 exit 代码是一个坏想法（它使程序逻辑较难理解），但是它在这个脚本中起作用。

总结归纳

在这一章中，我们向着程序交互性迈出了第一步；允许用户通过键盘向程序输入数据。使用目前 已经学过的技巧，有可能编写许多有用的程序，比如说特定的计算程序和容易使用的命令行工具 前端。在下一章中，我们将继续建立菜单驱动程序概念，让它更完善。

友情提示

仔细研究本章中的程序，并对程序的逻辑结构有一个完整的理解，这是非常重要的，因为即将到来的 程序会日益复杂。作为练习，用 test 命令而不是 `[[ ]]` 复合命令来重新编写本章中的程序。 提示：使用 grep 命令来计算正则表达式及其退出状态。这会是一个不错的实践。

拓展阅读

- Bash 参考手册有一章关于内部命令的内容，其包括了 read 命令：

  http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins

流程控制：while/until 循环

在前面的章节中，我们开发了菜单驱动程序，来产生各种各样的系统信息。虽然程序能够运行， 但它仍然存在重大的可用问题。它只能执行单一的选择，然后终止。更糟糕地是，如果做了一个 无效的选择，程序会以错误终止，而没有给用户提供再试一次的机会。如果我们能构建程序， 以致于程序能够重复显示菜单，而且能一次由一次的选择，直到用户选择退出程序，这样的程序会更好一些。

在这一章中，我们将看一个叫做循环的程序概念，其可用来使程序的某些部分重复。shell 为循环提供了三个复合命令。 本章我们将查看其中的两个命令，随后章节介绍第三个命令。

循环

日常生活中充满了重复性的活动。每天去散步，遛狗，切胡萝卜，所有任务都要重复一系列的步骤。 让我们以切胡萝卜为例。如果我们用伪码表达这种活动，它可能看起来像这样：

1. 准备切菜板
2. 准备菜刀
3. 把胡萝卜放到切菜板上
4. 提起菜刀
5. 向前推进胡萝卜
6. 切胡萝卜
7. 如果切完整个胡萝卜，就退出，要不然回到第四步继续执行

从第四步到第七步形成一个循环。重复执行循环内的动作直到满足条件“切完整个胡萝卜”。

bash 能够表达相似的想法。比方说我们想要按照顺序从1到5显示五个数字。可如下构造一个 bash 脚本：

```
#!/bin/bash
# while-count: display a series of numbers
count=1
while [ $count -le 5 ]; do
    echo $count
    count=$((count + 1))
done
echo "Finished."
```

当执行的时候，这个脚本显示如下信息：

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

while 命令的语法是：

```
while commands; do commands; done
```

和 if 一样， while 计算一系列命令的退出状态。只要退出状态为零，它就执行循环内的命令。 在上面的脚本中，创建了变量 count ，并初始化为1。 while 命令将会计算 test 命令的退出状态。 只要 test 命令返回退出状态零，循环内的所有命令就会

执行。每次循环结束之后，会重复执行 test 命令。 第六次循环之后， count 的数值增加到6， test 命令不再返回退出状态零，且循环终止。 程序继续执行循环之后的语句。

我们可以使用一个 while 循环，来提高前面章节的 read-menu 程序：

```
#!/bin/bash
# while-menu: a menu driven system information program
DELAY=3 # Number of seconds to display results
while [[ $REPLY != 0 ]]; do
    clear
    cat <<- _EOF_
        Please Select:
        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
    _EOF_
    read -p "Enter selection [0-3] > "
    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        sleep $DELAY
        fi
    if [[ $REPLY == 2 ]]; then
        df -h
        sleep $DELAY
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
    sleep $DELAY
    fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

通过把菜单包含在 while 循环中，每次用户选择之后，我们能够让程序重复显示菜单。只要 REPLY 不 等于"0"，循环就会继续，菜单就能显示，从而用户有机会重新选择。每次动作完成之后，会执行一个 sleep 命令，所以在清空屏幕和重新显示菜单之前，程序将会停顿几秒钟，为的是能够看到选项输出结果。 一旦 REPLY 等于“0”，则表示选择了“退出”选项，循环就会终止，程序继续执行 done 语句之后的代码。

跳出循环

bash 提供了两个内部命令，它们可以用来在循环内部控制程序流程。这个 break 命令立即终止一个循环， 且程序继续执行循环之后的语句。这个 continue 命令导致程序跳过循环中剩余的语句，且程序继续执行 下一次循环。这里我们看看采用了 break 和 continue 两个命令的 while-menu 程序版本：

```
#!/bin/bash
# while-menu2: a menu driven system information program
DELAY=3 # Number of seconds to display results
while true; do
    clear
    cat <<- _EOF_
        Please Select:
        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
    _EOF_
    read -p "Enter selection [0-3] > "
    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        sleep $DELAY
        continue
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        sleep $DELAY
        continue
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        sleep $DELAY
        continue
    fi
    if [[ $REPLY == 0 ]]; then
        break
    fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

在这个脚本版本中，我们设置了一个无限循环（就是自己永远不会终止的循环），通过使用 true 命令 为 while 提供一个退出状态。因为 true 的退出状态总是为零，所以循环永远不会终止。这是一个 令人惊讶的通用脚本编程技巧。因为循环自己永远不会结束，所以由程序员在恰当的时候提供某种方法来跳出循环。 此脚本，当选择"0"选项的时候，break 命令被用来退出循环。continue 命令被包含在其它选择动作的末尾， 为的是更加高效执行。通过使用 continue 命令，当一个选项确定后，程序会跳过不需要的代码。例如， 如果选择了选项"1"，则没有理由去测试其它选项。

这个 until 命令与 while 非常相似，除了当遇到一个非零退出状态的时候， while 退出循环， 而 until 不退出。一个 until 循环会继续执行直到它接受了一个退出状态零。在我们的 while-count 脚本中， 我们继续执行循环直到 count 变量的数值小于或等于5。我们可以得到相同的结果，通过在脚本中使用 until 命令：

```
#!/bin/bash
# until-count: display a series of numbers
count=1
until [ $count -gt 5 ]; do
    echo $count
    count=$((count + 1))
done
echo "Finished."
```

通过把 test 表达式更改为 $count -gt 5 ， until 会在正确的时间终止循环。决定使用 while 循环 还是 until 循环，通常是选择一个 test 可以编写地很清楚的循环。

使用循环读取文件

while 和 until 能够处理标准输入。这就可以使用 while 和 until 处理文件。在下面的例子中， 我们将显示在前面章节中使用的 distros.txt 文件的内容：

```
#!/bin/bash
# while-read: read lines from a file
while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
        $distro \
        $version \
        $release
done < distros.txt
```

为了重定向文件到循环中，我们把重定向操作符放置到 done 语句之后。循环将使用 read 从重定向文件中读取 字段。这个 read 命令读取每个文本行之后，将会退出，其退出状态为零，直到到达文件末尾。到时候，它的 退出状态为非零数值，因此终止循环。也有可能把标准输入管道到循环中。

```
#!/bin/bash
# while-read2: read lines from a file
sort -k 1,1 -k 2n distros.txt | while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
        $distro \
        $version \
        $release
done
```

这里我们接受 sort 命令的标准输出，然后显示文本流。然而，因为管道将会在子 shell 中执行 循环，当循环终止的时候，循环中创建的任意变量或赋值的变量都会消失，记住这一点很重要。

总结归纳

通过引入循环，和我们之前遇到的分支，子例程和序列，我们已经介绍了程序流程控制的主要类型。 bash 还有一些锦囊妙计，但它们都是关于这些基本概念的完善。

拓展阅读

- Linux 文档工程中的 Bash 初学者指南一书中介绍了更多的 while 循环实例：

  http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html

- Wikipedia 中有一篇关于循环的文章，其是一篇比较长的关于流程控制的文章中的一部分：

  http://en.wikipedia.org/wiki/Control_flow#Loops

疑难排解

随着我们的脚本变得越来越复杂，当脚本运行错误，执行结果出人意料的时候, 我们就应该查看一下原因了。 在这一章中，我们将会看一些脚本中出现地常见错误类型，同时还会介绍几个可以跟踪和消除问题的有用技巧。

语法错误

一个普通的错误类型是语法。语法错误涉及到一些 shell 语法元素的拼写错误。大多数情况下，这类错误 会导致 shell 拒绝执行此脚本。

在以下讨论中，我们将使用下面这个脚本，来说明常见的错误类型：

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

参看脚本内容，我们知道这个脚本执行成功了：

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

丢失引号

如果我们编辑我们的脚本，并从跟随第一个 echo 命令的参数中，删除其末尾的双引号：

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
    echo "Number is equal to 1.
else
    echo "Number is not equal to 1."
fi
```

观察发生了什么：

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for
matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

这个脚本产生了两个错误。有趣地是，所报告的行号不是引号被删除的地方，而是程序中后面的文本行。 我们能知道为什么，如果我们跟随丢失引号文本行之后的程序。bash 会继续寻找右引号，直到它找到一个， 其就是这个紧随第二个 echo 命令之后的引号。找到这个引号之后，bash 变得很困惑，并且 if 命令的语法 被破坏了，因为现在这个 fi 语句在一个用引号引起来的（但是开放的）字符串里面。

在冗长的脚本中，此类错误很难找到。使用带有语法高亮的编辑器将会帮助查找错误。如果安装了 vim 的完整版， 通过输入下面的命令，可以使语法高亮生效：

```
:syntax on
```

丢失或意外的标记

另一个常见错误是忘记补全一个复合命令，比如说 if 或者是 while。让我们看一下，如果 我们删除 if 命令中测试之后的分号，会出现什么情况：

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ] then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

结果是这样的：

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token
`else'
/home/me/bin/trouble: line 9: `else'
```

再次，错误信息指向一个错误，其出现的位置靠后于实际问题所在的文本行。所发生的事情真是相当有意思。我们记得， if 能够接受一系列命令，并且会计算列表中最后一个命令的退出代码。在我们的程序中，我们打算这个列表由 单个命令组成，即 [，测试的同义词。这个 [命令把它后面的东西看作是一个参数列表。在我们这种情况下， 有三个参数：$number，=，和]。由于删除了分号，单词 then 被添加到参数列表中，从语法上讲， 这是合法的。随后的 echo 命令也是合法的。它被解释为命令列表中的另一个命令，if 将会计算命令的 退出代码。接下来遇到单词 else，但是它出局了，因为 shell 把它认定为一个 保留字（对于 shell 有特殊含义的单词），而不是一个命令名，因此报告错误信息。

预料不到的展开

可能有这样的错误，它们仅会间歇性地出现在一个脚本中。有时候这个脚本执行正常，其它时间会失败， 这是因为展开结果造成的。如果我们归还我们丢掉的分号，并把 number 的数值更改为一个空变量，我们 可以示范一下：

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

运行这个做了修改的脚本，得到以下输出：

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
Number is not equal to 1.
```

我们得到一个相当神秘的错误信息，其后是第二个 echo 命令的输出结果。这问题是由于 test 命令中 number 变量的展开结果造成的。当此命令：

```
[ $number = 1 ]
```

经过展开之后，number 变为空值，结果就是这样：

```
[  = 1 ]
```

这是无效的，所以就产生了错误。这个 = 操作符是一个二元操作符（它要求每边都有一个数值），但是第一个数值是缺失的，这样 test 命令就期望用一个一元操作符（比如 -z）来代替。进一步说，因为 test 命令运行失败了（由于错误），这个 if 命令接收到一个非零退出代码，因此执行第二个 echo 命令。

通过为 test 命令中的第一个参数添加双引号，可以更正这个问题：

```
[ "$number" = 1 ]
```

然后当展开操作发生地时候，执行结果将会是这样：

```
[ "" = 1 ]
```

其得到了正确的参数个数。除了代表空字符串之外，引号应该被用于这样的场合，一个要展开 成多单词字符串的数值，及其包含嵌入式空格的文件名。

逻辑错误

不同于语法错误，逻辑错误不会阻止脚本执行。虽然脚本会正常运行，但是它不会产生期望的结果，归咎于脚本的逻辑问题。虽然有不计其数的可能的逻辑错误，但下面是一些在脚本中找到的最常见的 逻辑错误类型：

1. 不正确的条件表达式。很容易编写一个错误的 if/then/else 语句，并且执行错误的逻辑。有时候逻辑会被颠倒，或者是逻辑结构不完整。

2. “超出一个值”错误。当编写带有计数器的循环语句的时候，为了计数在恰当的点结束，循环语句 可能要求从 0 开始计数，而不是从 1 开始，这有可能会被忽视。这些类型的错误要不导致循环计数太多，而“超出范围”，要不就是过早的结束了一次迭代，从而错过了最后一次迭代循环。

3. 意外情况。大多数逻辑错误来自于程序碰到了程序员没有预见到的数据或者情况。这也 可以包括出乎意料的展开，比如说一个包含嵌入式空格的文件名展开成多个命令参数而不是单个的文件名。

防错编程

当编程的时候，验证假设非常重要。这意味着要仔细得计算脚本所使用的程序和命令的退出状态。这里有个实例，基于一个真实的故事。为了在一台重要的服务器中执行维护任务，一位不幸的系统管理员写了一个脚本。这个脚本包含下面两行代码：

```
cd $dir_name
rm *
```

从本质上来说，这两行代码没有任何问题，只要是变量 dir_name 中存储的目录名字存在就可以。但是如果不是这样会发生什么事情呢？在那种情况下，cd 命令会运行失败，脚本会继续执行下一行代码，将会删除当前工作目录中的所有文件。完成不是期望的结果！由于这种设计策略，这个倒霉的管理员销毁了服务器中的一个重要部分。

让我们看一些能够提高这个设计的方法。首先，在 cd 命令执行成功之后，再运行 rm 命令，可能是明智的选择。

```
cd $dir_name && rm *
```

这样，如果 cd 命令运行失败后，rm 命令将不会执行。这样比较好，但是仍然有可能未设置变量 dir_name 或其变量值为空，从而导致删除了用户主目录下面的所有文件。这个问题也能够避免，通过检验变量 dir_name 中包含的目录名是否真正地存在：

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

通常，当某种情况（比如上述问题）发生的时候，最好是终止脚本执行，并对这种情况提示错误信息：

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
```

这里，我们检验了两种情况，一个名字，看看它是否为一个真正存在的目录，另一个是 cd 命令是否执行成功。如果任一种情况失败，就会发送一个错误说明信息到标准错误，然后脚本终止执行，并用退出状态 1 表明脚本执行失败。

验证输入

一个良好的编程习惯是如果一个程序可以接受输入数据，那么这个程序必须能够应对它所接受的任意数据。这 通常意味着必须非常仔细地筛选输入数据，以确保只有有效的输入数据才能被程序用来做进一步地处理。在前面章节 中我们学习 read 命令的时候，我们遇到过一个这样的例子。一个脚本中包含了下面一条测试语句，用来验证一个选择菜单：

```
[[ $REPLY =~ ^[0-3]$ ]]
```

这条测试语句非常明确。只有当用户输入是一个位于 0 到 3 范围内（包括 0 和 3）的数字的时候，这条语句才返回一个 0 退出状态。而其它任何输入概不接受。有时候编写这类测试条件非常具有挑战性，但是为了能产出一个高质量的脚本，付出还是必要的。

> 设计是时间的函数
>
> 当我还是一名大学生，在学习工业设计的时候，一位明智的教授说过一个项目的设计程度是由 给定设计师的时间量来决定的。如果给你五分钟来设计一款能够"杀死苍蝇"的产品，你会设计出一个苍蝇拍。如果给你五个月的时间，你可能会制作出激光制导的"反苍蝇系统"。
>
> 同样的原理适用于编程。有时候一个"快速但粗糙"的脚本就可以解决问题，但这个脚本只能被其作者使用一次。这类脚本很常见，为了节省气力也应该被快速地开发出来。所以这些脚本不需要太多的注释和防错检查。相反，如果一个脚本打算用于生产使用，也就是说，某个重要任务或者多个客户会不断地用到它，此时这个脚本就需要非常谨慎小心地开发了。

测试

在各类软件开发中（包括脚本），测试是一个重要的环节。在开源世界中有一句谚语，"早发布，常发布"，这句谚语就反映出这个事实（测试的重要性）。通过提早和经常发布，软件能够得到更多曝光去使用和测试。经验表明如果在开发周期的早期发现 bug，那么这些 bug 就越容易定位，而且越能低成本 的修复。

在之前的讨论中，我们知道了如何使用 stubs 来验证程序流程。在脚本开发的最初阶段，它们是一项有价值的技术 来检测我们的工作进度。

让我们看一下上面的文件删除问题，为了轻松测试，看看如何修改这些代码。测试原本那个代码片段将是危险的，因为它的目的是要删除文件，但是我们可以修改代码，让测试安全：

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        echo rm * # TESTING
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
exit # TESTING
```

因为在满足出错条件的情况下代码可以打印出有用信息，所以我们没有必要再添加任何额外信息了。 最重要的改动是仅在 rm 命令之前放置了一个 echo 命令， 为的是把 rm 命令及其展开的参数列表打印出来，而不是执行实际的 rm 命令语句。这个改动可以安全的执行代码。 在这段代码的末尾，我们放置了一个 exit 命令来结束测试，从而防止执行脚本其它部分的代码。 这个需求会因脚本的设计不同而变化。

我们也在代码中添加了一些注释，用来标记与测试相关的改动。当测试完成之后，这些注释可以帮助我们找到并删除所有的更改。

测试案例

为了执行有用的测试，开发和使用好的测试案例是很重要的。这个要求可以通过谨慎地选择输入数据或者运行边缘案例和极端案例来完成。 在我们的代码片段中（是非常简单的代码），我们想要知道在下面的三种具体情况下这段代码是怎样执行的：

1. dir_name 包含一个已经存在的目录的名字
2. dir_name 包含一个不存在的目录的名字
3. dir_name 为空

通过执行以上每一个测试条件，就达到了一个良好的测试覆盖率。

正如设计，测试也是一个时间的函数。不是每一个脚本功能都需要做大量的测试。问题关键是确定什么功能是最重要的。因为 测试若发生故障会存在如此潜在的破坏性，所以我们的代码片在设计和测试段期间都应值得仔细推敲。

调试

如果测试暴露了脚本中的一个问题，那下一步就是调试了。“一个问题”通常意味着在某种情况下，这个脚本的执行 结果不是程序员所期望的结果。若是这种情况，我们需要仔细确认这个脚本实际到底要完成什么任务，和为什么要这样做。 有时候查找 bug 要牵涉到许多监测工作。一个设计良好的脚本会对查找错误有帮助。设计良好的脚本应该具备防卫能力， 能够监测异常条件，并能为用户提供有用的反馈信息。 然而有时候，出现的问题相当稀奇，出人意料，这时候就需要更多的调试技巧了。

找到问题区域

在一些脚本中，尤其是一些代码比较长的脚本，有时候隔离脚本中与出现的问题相关的代码区域对查找问题很有效。 隔离的代码区域并不总是真正的错误所在，但是隔离往往可以深入了解实际的错误原因。可以用来隔离代码的一项 技巧是“添加注释”。例如，我们的文件删除代码可以修改成这样，从而决定注释掉的这部分代码是否导致了一个错误：

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
# else
#
    echo "no such directory: '$dir_name'" >&2
#
    exit 1
fi
```

通过给脚本中的一个逻辑区块内的每条语句的开头添加一个注释符号，我们就阻止了这部分代码的执行。然后可以再次执行测试， 来看看清除的代码是否影响了错误的行为。

追踪

在一个脚本中，错误往往是由意想不到的逻辑流导致的。也就是说，脚本中的一部分代码或者从未执行，或是以错误的顺序， 或在错误的时间给执行了。为了查看真实的程序流，我们使用一项叫做追踪（tracing）的技术。

一种追踪方法涉及到在脚本中添加可以显示程序执行位置的提示性信息。我们可以添加提示信息到我们的代码片段中：

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
echo "deleting files" >&2
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
echo "file deletion complete" >&2
```

我们把提示信息输出到标准错误输出，让其从标准输出中分离出来。我们也没有缩进包含提示信息的语句，这样 想要删除它们的时候，能比较容易找到它们。

当这个脚本执行的时候，就可能看到文件删除操作已经完成了：

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

bash 还提供了一种名为追踪的方法，这种方法可通过 -x 选项和 set 命令加上 -x 选项两种途径实现。 拿我们之前的 trouble 脚本为例，给该脚本的第一行语句添加 -x 选项，我们就能追踪整个脚本。

```
#!/bin/bash -x
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

当脚本执行后，输出结果看起来像这样:

```
[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

追踪生效后，我们看到脚本命令展开后才执行。行首的加号表明追踪的迹象，使其与常规输出结果区分开来。 加号是追踪输出的默认字符。它包含在 PS4（提示符4）shell 变量中。可以调整这个变量值让提示信息更有意义。 这里，我们修改该变量的内容，让其包含脚本中追踪执行到的当前行的行号。注意这里必须使用单引号是为了防止变量展开，直到 提示符真正使用的时候，就不需要了。

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

我们可以使用 set 命令加上 -x 选项，为脚本中的一块选择区域，而不是整个脚本启用追踪。

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

我们使用 set 命令加上 -x 选项来启动追踪，+x 选项关闭追踪。这种技术可以用来检查一个有错误的脚本的多个部分。

执行时检查数值

伴随着追踪，在脚本执行的时候显示变量的内容，以此知道脚本内部的工作状态，往往是很用的。 使用额外的 echo 语句通常会奏效。

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

在这个简单的示例中，我们只是显示变量 number 的数值，并为其添加注释，随后利于其识别和清除。 当查看脚本中的循环和算术语句的时候，这种技术特别有用。

总结

在这一章中，我们仅仅看了几个在脚本开发期间会出现的问题。当然，还有很多。这章中描述的技术对查找 大多数的常见错误是有效的。调试是一种艺术，可以通过开发经验，在知道如何避免错误(整个开发过程中不断测试) 以及在查找 bug（有效利用追踪）两方面都会得到提升。

拓展阅读

- Wikipedia 上面有两篇关于语法和逻辑错误的短文：

  http://en.wikipedia.org/wiki/Syntax_error

  http://en.wikipedia.org/wiki/logic_error

- 网上有很多关于技术层面的 bash 编程的资源：

  http://mywiki.wooledge.org/BashPitfalls

  http://tldp.org/LDP/abs/html/gotchas.html

  http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html

- 想要学习从编写良好的 Unix 程序中得知的基本概念，可以参考 Eric Raymond 的《Unix 编程的艺术》这本 伟大的著作。书中的许多想法都能适用于 shell 脚本：

  http://www.faqs.org/docs/artu/

  http://www.faqs.org/docs/artu/ch01s06.html

- 对于真正的高强度的调试，参考这个 Bash Debugger：

  http://bashdb.sourceforge.net/

流程控制：case 分支

在这一章中，我们将继续看一下程序的流程控制。在第28章中，我们构建了一些简单的菜单并创建了用来 应对各种用户选择的程序逻辑。为此，我们使用了一系列的 if 命令来识别哪一个可能的选项已经被选中。 这种类型的构造经常出现在程序中，出现频率如此之多，以至于许多编程语言（包括 shell） 专门为多选决策提供了一种流程控制机制。

case

Bash 的多选复合命令称为 case。它的语法规则如下所示：

```
case word in
    [pattern [| pattern]...) commands ;;]...
esac
```

如果我们看一下第28章中的读菜单程序，我们就知道了用来应对一个用户选项的逻辑流程：

```
#!/bin/bash
# read-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi
```

使用 case 语句，我们可以用更简单的代码替换这种逻辑：

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
case $REPLY in
    0)  echo "Program terminated."
        exit
        ;;
    1)  echo "Hostname: $HOSTNAME"
        uptime
        ;;
    2)  df -h
        ;;
    3)  if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        ;;
    *)  echo "Invalid entry" >&2
        exit 1
        ;;
esac
```

case 命令检查一个变量值，在我们这个例子中，就是 REPLY 变量的变量值，然后试图去匹配其中一个具体的模式。 当与之相匹配的模式找到之后，就会执行与该模式相关联的命令。若找到一个模式之后，就不会再继续寻找。

模式

这里 case 语句使用的模式和路径展开中使用的那些是一样的。模式以一个 “)” 为终止符。这里是一些有效的模式。

表 32-1：case 模式实例

模式	描述
a)	若单词为 “a”，则匹配
[[:alpha:]])	若单词是一个字母字符，则匹配
???)	若单词只有3个字符，则匹配
.txt)	若单词以 “.txt” 字符结尾，则匹配
)	匹配任意单词。把这个模式做为 case 命令的最后一个模式，是一个很好的做法， 可以捕捉到任意一个与先前模式不匹配的数值；也就是说，捕捉到任何可能的无效值。

这里是一个模式使用实例：

```
#!/bin/bash
read -p "enter word > "
case $REPLY in
    [[:alpha:]])        echo "is a single alphabetic character." ;;
    [ABC][0-9])         echo "is A, B, or C followed by a digit." ;;
    ???)                echo "is three characters long." ;;
    *.txt)              echo "is a word ending in '.txt'" ;;
    *)                  echo "is something else." ;;
esac
```

还可以使用竖线字符作为分隔符，把多个模式结合起来。这就创建了一个 “或” 条件模式。这对于处理诸如大小写字符很有用处。例如：

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "
case $REPLY in
q|Q) echo "Program terminated."
    exit
    ;;
a|A) echo "Hostname: $HOSTNAME"
    uptime
    ;;
b|B) df -h
    ;;
c|C) if [[ $(id -u) -eq 0 ]]; then
        echo "Home Space Utilization (All Users)"
        du -sh /home/*
    else
        echo "Home Space Utilization ($USER)"
        du -sh $HOME
    fi
    ;;
*)   echo "Invalid entry" >&2
    exit 1
    ;;
esac
```

这里，我们更改了 case-menu 程序的代码，用字母来代替数字做为菜单选项。注意新模式如何使得大小写字母都是有效的输入选项。

执行多个动作

早于版本号4.0的 bash，case 语法只允许执行与一个成功匹配的模式相关联的动作。 匹配成功之后，命令将会终止。这里我们看一个测试一个字符的脚本：

```
#!/bin/bash
# case4-1: test a character
read -n 1 -p "Type a character > "
echo
case $REPLY in
   [[:upper:]])    echo "'$REPLY' is upper case." ;;
   [[:lower:]])    echo "'$REPLY' is lower case." ;;
   [[:alpha:]])    echo "'$REPLY' is alphabetic." ;;
   [[:digit:]])    echo "'$REPLY' is a digit." ;;
   [[:graph:]])    echo "'$REPLY' is a visible character." ;;
   [[:punct:]])    echo "'$REPLY' is a punctuation symbol." ;;
   [[:space:]])    echo "'$REPLY' is a whitespace character." ;;
   [[:xdigit:]])   echo "'$REPLY' is a hexadecimal digit." ;;
esac
```

运行这个脚本，输出这些内容：

```
[me@linuxbox ~]$ case4-1
Type a character > a
'a' is lower case.
```

大多数情况下这个脚本工作是正常的，但若输入的字符不止与一个 POSIX 字符集匹配的话，这时脚本就会出错。 例如，字符 “a” 既是小写字母，也是一个十六进制的数字。早于4.0的 bash，对于 case 语法绝不能匹配 多个测试条件。现在的 bash 版本，添加 “;;&” 表达式来终止每个行动，所以现在我们可以做到这一点：

```
#!/bin/bash
# case4-2: test a character
read -n 1 -p "Type a character > "
echo
case $REPLY in
    [[:upper:]])    echo "'$REPLY' is upper case." ;;&
    [[:lower:]])    echo "'$REPLY' is lower case." ;;&
    [[:alpha:]])    echo "'$REPLY' is alphabetic." ;;&
    [[:digit:]])    echo "'$REPLY' is a digit." ;;&
    [[:graph:]])    echo "'$REPLY' is a visible character." ;;&
    [[:punct:]])    echo "'$REPLY' is a punctuation symbol." ;;&
    [[:space:]])    echo "'$REPLY' is a whitespace character." ;;&
    [[:xdigit:]])   echo "'$REPLY' is a hexadecimal digit." ;;&
esac
```

当我们运行这个脚本的时候，我们得到这些：

```
[me@linuxbox ~]$ case4-2
Type a character > a
'a' is lower case.
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```

添加的 “;;&” 的语法允许 case 语句继续执行下一条测试，而不是简单地终止运行。

总结

case 命令是我们编程技巧口袋中的一个便捷工具。在下一章中我们将看到，对于处理某些类型的问题来说，case 命令是一个完美的工具。

拓展阅读

- Bash 参考手册的条件构造一节详尽的介绍了 case 命令：

  http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21

- 高级 Bash 脚本指南提供了更深一层的 case 应用实例：

  http://tldp.org/LDP/abs/html/testbranch.html

位置参数

现在我们的程序还缺少一种本领，就是接收和处理命令行选项和参数的能力。在这一章中，我们将探究一些能 让程序访问命令行内容的 shell 性能。

访问命令行

shell 提供了一个称为位置参数的变量集合，这个集合包含了命令行中所有独立的单词。这些变量按照从0到9给予命名。 可以以这种方式讲明白：

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

一个非常简单的脚本，显示从 $0 到 $9 所有变量的值。当不带命令行参数执行该脚本时，输出结果如下：

```
[me@linuxbox ~]$ posit-param
$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

即使不带命令行参数，位置参数 $0 总会包含命令行中出现的第一个单词，也就是已执行程序的路径名。 当带参数执行脚本时，我们看看输出结果：

```
[me@linuxbox ~]$ posit-param a b c d
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

注意： 实际上通过参数展开方式你可以访问的参数个数多于9个。只要指定一个大于9的数字，用花括号把该数字括起来就可以。 例如 ${10}， ${55}， ${211}，等等。

确定参数个数

另外 shell 还提供了一个名为 $#，可以得到命令行参数个数的变量:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

结果是：

```
[me@linuxbox ~]$ posit-param a b c d
Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

shift - 访问多个参数的利器

但是如果我们给一个程序添加大量的命令行参数，会怎么样呢？ 正如下面的例子：

```
[me@linuxbox ~]$ posit-param *
Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

在这个例子运行的环境下，通配符 * 展开成82个参数。我们如何处理那么多的参数？ 为此，shell 提供了一种方法，尽管笨拙，但可以解决这个问题。执行一次 shift 命令， 就会导致所有的位置参数 "向下移动一个位置"。事实上，用 shift 命令也可以 处理只有一个参数的情况（除了其值永远不会改变的变量 $0）：

```
#!/bin/bash
# posit-param2: script to display all arguments
count=1
while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done
```

每次 shift 命令执行的时候，变量 $2 的值会移动到变量 $1 中，变量 $3 的值会移动到变量 $2 中，依次类推。 变量 $# 的值也会相应的减1。

在该 posit-param2 程序中，我们编写了一个计算剩余参数数量，只要参数个数不为零就会继续执行的 while 循环。 我们显

示当前的位置参数，每次循环迭代变量 count 的值都会加1，用来计数处理的参数数量， 最后，执行 shift 命令加载 $1，其值为下一个位置参数的值。这里是程序运行后的输出结果:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

简单应用

即使没有 shift 命令，也可以用位置参数编写一个有用的应用。举例说明，这里是一个简单的输出文件信息的程序：

```
#!/bin/bash
# file_info: simple file information program
PROGNAME=$(basename $0)
if [[ -e $1 ]]; then
    echo -e "\nFile Type:"
    file $1
    echo -e "\nFile Status:"
    stat $1
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi
```

这个程序显示一个具体文件的文件类型（由 file 命令确定）和文件状态（来自 stat 命令）。该程序一个有意思 的特点是 PROGNAME 变量。它的值就是 basename $0 命令的执行结果。这个 basename 命令清除 一个路径名的开头部分，只留下一个文件的基本名称。在我们的程序中，basename 命令清除了包含在 $0 位置参数 中的路径名的开头部分，$0 中包含着我们示例程序的完整路径名。当构建提示信息正如程序结尾的使用信息的时候， basename $0 的执行结果就很有用处。按照这种方式编码，可以重命名该脚本，且程序信息会自动调整为 包含相应的程序名称。

Shell 函数中使用位置参数

正如位置参数被用来给 shell 脚本传递参数一样，它们也能够被用来给 shell 函数传递参数。为了说明这一点， 我们将把 file_info 脚本转变成一个 shell 函数：

```
file_info () {
  # file_info: function to display file information
  if [[ -e $1 ]]; then
      echo -e "\nFile Type:"
      file $1
      echo -e "\nFile Status:"
      stat $1
  else
      echo "$FUNCNAME: usage: $FUNCNAME file" >&2
      return 1
  fi
}
```

现在，如果一个包含 shell 函数 file_info 的脚本调用该函数，且带有一个文件名参数，那这个参数会传递给 file_info 函数。

通过此功能，我们可以写出许多有用的 shell 函数，这些函数不仅能在脚本中使用，也可以用在 .bashrc 文件中。

注意那个 PROGNAME 变量已经改成 shell 变量 FUNCNAME 了。shell 会自动更新 FUNCNAME 变量，以便 跟踪当前执行的 shell 函数。注意位置参数 $0 总是包含命令行中第一项的完整路径名（例如，该程序的名字）， 但不会包含这个我们可能期望的 shell 函数的名字。

处理集体位置参数

有时候把所有的位置参数作为一个集体来管理是很有用的。例如，我们可能想为另一个程序编写一个 “包裹程序”。 这意味着

我们会创建一个脚本或 shell 函数，来简化另一个程序的执行。包裹程序提供了一个神秘的命令行选项 列表，然后把这个参数列表传递给下一级的程序。

为此 shell 提供了两种特殊的参数。他们二者都能扩展成完整的位置参数列表，但以相当微妙的方式略有不同。它们是：

表 32-1: 和 @ 特殊参数

参数	描述
$	展开成一个从1开始的位置参数列表。当它被用双引号引起来的时候，展开成一个由双引号引起来 的字符串，包含了所有的位置参数，每个位置参数由 shell 变量 IFS 的第一个字符（默认为一个空格）分隔开。
$@	展开成一个从1开始的位置参数列表。当它被用双引号引起来的时候， 它把每一个位置参数展开成一个由双引号引起来的分开的字符串。

下面这个脚本用程序中展示了这些特殊参数：

```
#!/bin/bash
# posit-params3 : script to demonstrate $* and $@
print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}
pass_params () {
    echo -e "\n" '$* :';      print_params   $*
    echo -e "\n" '"$*" :';    print_params   "$*"
    echo -e "\n" '$@ :';      print_params   $@
    echo -e "\n" '"$@" :';    print_params   "$@"
}
pass_params "word" "words with spaces"
```

在这个相当复杂的程序中，我们创建了两个参数：“word” 和 “words with spaces”，然后把它们 传递给 pass_params 函数。这个函数，依次，再把两个参数传递给 print_params 函数， 使用了特殊参数 $* 和 $@ 提供的四种可用方法。脚本运行后，揭示了这两个特殊参数存在的差异：

```
[me@linuxbox ~]$ posit-param3
 $* :
$1 = word
$2 = words
$3 = with
$4 = spaces
 "$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =
 $@ :
$1 = word
$2 = words
$3 = with
$4 = spaces
 "$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

通过我们的参数，$* 和 $@ 两个都产生了一个有四个词的结果：

```
word words with spaces
"$*" produces a one word result:
    "word words with spaces"
"$@" produces a two word result:
    "word" "words with spaces"
```

这个结果符合我们实际的期望。我们从中得到的教训是尽管 shell 提供了四种不同的得到位置参数列表的方法， 但到目前为止， "$@" 在大多数情况下是最有用的方法，因为它保留了每一个位置参数的完整性。

一个更复杂的应用

经过长时间的间断，我们将恢复程序 sys_info_page 的工作。我们下一步要给程序添加如下几个命令行选项：

- 输出文件。 我们将添加一个选项，以便指定一个文件名，来包含程序的输出结果。 选项格式要么是 -f file，要么是 --file file

- 交互模式。这个选项将提示用户输入一个输出文件名，然后判断是否指定的文件已经存在了。如果文件存在， 在覆盖这个存在的文件之前会提示用户。这个选项可以通过 -i 或者 --interactive 来指定。

- 帮助。指定 -h 选项 或者是 --help 选项，可导致程序输出提示性的使用信息。

这里是处理命令行选项所需的代码：

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}
# process command line options
interactive=
filename=
while [[ -n $1 ]]; do
    case $1 in
    -f | --file)            shift
                            filename=$1
                            ;;
    -i | --interactive)     interactive=1
                            ;;
    -h | --help)            usage
                            exit
                            ;;
    *)                      usage >&2
                            exit 1
                            ;;
    esac
    shift
done
```

首先，我们添加了一个叫做 usage 的 shell 函数，以便显示帮助信息，当启用帮助选项或敲写了一个未知选项的时候。

下一步，我们开始处理循环。当位置参数 $1 不为空的时候，这个循环会持续运行。在循环的底部，有一个 shift 命令， 用来提升位置参数，以便确保该循环最终会终止。在循环体内，我们使用了一个 case 语句来检查当前位置参数的值， 看看它是否匹配某个支持的选项。若找到了匹配项，就会执行与之对应的代码。若没有，就会打印出程序使用信息， 该脚本终止且执行错误。

处理 -f 参数的方式很有意思。当监测到 -f 参数的时候，会执行一次 shift 命令，从而提升位置参数 $1 为 伴随着 -f 选项的 filename 参数。

我们下一步添加代码来实现交互模式：

```
# interactive mode
if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
            Y|y)    break
                    ;;
            Q|q)    echo "Program terminated."
                    exit
                    ;;
            *)      continue
                    ;;
            esac
        elif [[ -z $filename ]]; then
            continue
        else
            break
        fi
    done
fi
```

若 interactive 变量不为空，就会启动一个无休止的循环，该循环包含文件名提示和随后存在的文件处理代码。 如果所需要的输出文件已经存在，则提示用户覆盖，选择另一个文件名，或者退出程序。如果用户选择覆盖一个 已经存在的文件，则会执行 break 命令终止循环。注意 case 语句是怎样只检测用户选择了覆盖还是退出选项。 其它任何选择都会导致循环继续并提示用户再次选择。

为了实现这个输出文件名的功能，首先我们必须把现有的这个写页面（page-writing）的代码转变成一个 shell 函数， 一会儿就会明白这样做的原因：

```
write_html_page () {
    cat <<- _EOF_
        <HTML>
            <HEAD>
                <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                <H1>$TITLE</H1>
                <P>$TIMESTAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </BODY>
        </HTML>
    _EOF_
    return
}
# output html page
if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

解决 -f 选项逻辑的代码出现在以上程序片段的末尾。在这段代码中，我们测试一个文件名是否存在，若文件名存在， 则执行另一个测试看看该文件是不是可写文件。为此，会运行 touch 命令， 紧随其后执行一个测试，来决定 touch 命令 创建的文件是否是个普通文件。这两个测试考虑到了输入是无效路径名（touch 命令执行失败），和一个普通文件已经存在的情况。

正如我们所看到的，程序调用 write_html_page 函数来生成实际的网页。函数输出要么直接定向到 标准输出（若 filename 变量为空的话）要么重定向到具体的文件中。

总结

伴随着位置参数的加入，现在我们能编写相当具有功能性的脚本。例如，重复性的任务，位置参数使得编写 非常有用的，可以放置在一个用户的 .bashrc 文件中的 shell 函数成为可能。

我们的 sys_info_page 程序日渐精进。这里是一个完整的程序清单，最新的更改用高亮显示：

```
#!/bin/bash
# sys_info_page: program to output a system information page
PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"
report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
    _EOF_
    return
}
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
    _EOF_
    return
}
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}
write_html_page () {
    cat <<- _EOF_
        <HTML>
            <HEAD>
                <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                <H1>$TITLE</H1>
                <P>$TIMESTAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </BODY>
        </HTML>
    _EOF_
    return
}
# process command line options
interactive=
filename=
while [[ -n $1 ]]; do
    case $1 in
        -f | --file)          shift
                              filename=$1
                              ;;
        -i | --interactive)   interactive=1
                              ;;
        -h | --help)          usage
                              exit
                              ;;
        *)                    usage >&2
                              exit 1
                              ;;
    esac
    shift
done
# interactive mode
```

```
# interactive mode
if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
                Y|y)    break
                        ;;
                Q|q)    echo "Program terminated."
                        exit
                        ;;
                *)      continue
                        ;;
            esac
        fi
    done
fi
# output html page
if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

我们还没有完成。仍然还有许多事情我们可以做，可以改进。

拓展阅读

- *Bash Hackers Wiki* 上有一篇不错的关于位置参数的文章：

  http://wiki.bash-hackers.org/scripting/posparams

- Bash 的参考手册有一篇关于特殊参数的文章，包括 $* 和 $@：

  http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters

- 除了本章讨论的技术之外，bash 还包含一个叫做 getopts 的内部命令，此命令也可以用来处理命令行参数。 bash 参考页面的 SHELL BUILTIN COMMANDS 一节介绍了这个命令，Bash Hackers Wiki 上也有对它的描述：

  http://wiki.bash-hackers.org/howto/getopts_tutorial

流程控制：for 循环

在这关于流程控制的最后一章中，我们将看看另一种 shell 循环构造。for 循环不同于 while 和 until 循环，因为 在循环中，它提供了一种处理序列的方式。这证明在编程时非常有用。因此在 bash 脚本中，for 循环是非常流行的构造。

实现一个 for 循环，很自然的，要用 for 命令。在现代版的 bash 中，有两种可用的 for 循环格式。

for: 传统 shell 格式

原来的 for 命令语法是：

```
for variable [in words]; do
    commands
done
```

这里的 variable 是一个变量的名字，这个变量在循环执行期间会增加，words 是一个可选的条目列表， 其值会按顺序赋值给 variable，commands 是在每次循环迭代中要执行的命令。

在命令行中 for 命令是很有用的。我们可以很容易的说明它是如何工作的：

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

在这个例子中，for 循环有一个四个单词的列表：“A”，“B”，“C”，和 “D”。由于这四个单词的列表，for 循环会执行四次。 每次循环执行的时候，就会有一个单词赋值给变量 i。在循环体内，我们有一个 echo 命令会显示 i 变量的值，来演示赋值结果。 正如 while 和 until 循环，done 关键字会关闭循环。

for 命令真正强大的功能是我们可以通过许多有趣的方式创建 words 列表。例如，通过花括号展开：

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

或者路径名展开：

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

或者命令替换：

```
#!/bin/bash
# longest-word : find longest string in a file
while [[ -n $1 ]]; do
    if [[ -r $1 ]]; then
        max_word=
        max_len=0
        for i in $(strings $1); do
            len=$(echo $i | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$i
            fi
        done
        echo "$1: '$max_word' ($max_len characters)"
    fi
    shift
done
```

在这个示例中，我们要在一个文件中查找最长的字符串。当在命令行中给出一个或多个文件名的时候， 该程序会使用 strings 程序（其包含在 GNU binutils 包中）， 为每一个文件产生一个可读的文本格式的 “words” 列表。 然后这个 for 循环依次处理每个单词，判断当前这个单词是否为目前为止找到的最长的一个。当循环结束的时候，显示出最长的单词。

如果省略掉 for 命令的可选项 words 部分，for 命令会默认处理位置参数。 我们将修改 longest-word 脚本，来使用这种方式：

```
#!/bin/bash
# longest-word2 : find longest string in a file
for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=0
        for j in $(strings $i); do
            len=$(echo $j | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

正如我们所看到的，我们已经更改了最外围的循环，用 for 循环来代替 while 循环。通过省略 for 命令的 words 列表， 用位置参数替而代之。在循环体内，之前的变量 i 已经改为变量 j。同时 shift 命令也被淘汰掉了。

> 为什么是 *i*？
>
> 你可能已经注意到上面所列举的 for 循环的实例都选择 i 作为变量。为什么呢？ 实际上没有具体原因，除了传统习惯。 for 循环使用的变量可以是任意有效的变量，但是 i 是最常用的一个，其次是 j 和 k。
>
> 这一传统的基础源于 Fortran 编程语言。在 Fortran 语言中，以字母 I，J，K，L 和 M 开头的未声明变量的类型 自动设为整形，而以其它字母开头的变量则为实数类型（带有小数的数字）。这种行为导致程序员使用变量 I，J，和 K 作为循环变量， 因为当需要一个临时变量（正如循环变量）的时候，使用它们工作量比较少。这也引出了如下基于 fortran 的俏皮话：
>
> “神是真实的，除非是声明的整数。”

for: C 语言格式

最新版本的 bash 已经添加了第二种格式的 for 命令语法，该语法相似于 C 语言中的 for 语法格式。 其它许多编程语言也支持这种格式：

```
for (( expression1; expression2; expression3 )); do
    commands
done
```

这里的 expression1，expression2，和 expression3 都是算术表达式，commands 是每次循环迭代时要执行的命令。 在行为方面，这相当于以下构造形式：

```
(( expression1 ))
while (( expression2 )); do
    commands
    (( expression3 ))
done
```

expression1 用来初始化循环条件，expression2 用来决定循环结束的时间，还有在每次循环迭代的末尾会执行 expression3。

这里是一个典型应用：

```
#!/bin/bash
# simple_counter : demo of C style for command
for (( i=0; i<5; i=i+1 )); do
    echo $i
done
```

脚本执行之后，产生如下输出：

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

在这个示例中，expression1 初始化变量 i 的值为0，expression2 允许循环继续执行只要变量 i 的值小于5， 还有每次循环迭代时，expression3 会把变量 i 的值加1。

C 语言格式的 for 循环对于需要一个数字序列的情况是很有用处的。我们将在接下来的两章中看到几个这样的应用实例。

总结

学习了 for 命令的知识，现在我们将对我们的 sys_info_page 脚本做最后的改进。 目前，这个 report_home_space 函数看起来像这样：

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
        <H2>Home Space Utilization (All Users)</H2>
        <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
        <H2>Home Space Utilization ($USER)</H2>
        <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

下一步，我们将重写它，以便提供每个用户主目录的更详尽信息，并且包含用户主目录中文件和目录的总个数：

```
report_home_space () {
    local format="%8s%10s%10s\n"
    local i dir_list total_files total_dirs total_size user_name
    if [[ $(id -u) -eq 0 ]]; then
        dir_list=/home/*
        user_name="All Users"
    else
        dir_list=$HOME
        user_name=$USER
    fi
    echo "<H2>Home Space Utilization ($user_name)</H2>"
    for i in $dir_list; do
        total_files=$(find $i -type f | wc -l)
        total_dirs=$(find $i -type d | wc -l)
        total_size=$(du -sh $i | cut -f 1)
        echo "<H3>$i</H3>"
        echo "<PRE>"
        printf "$format" "Dirs" "Files" "Size"
        printf "$format" "----" "-----" "----"
        printf "$format" $total_dirs $total_files $total_size
        echo "</PRE>"
    done
    return
}
```

这次重写应用了目前为止我们学过的许多知识。我们仍然测试超级用户（superuser），但是我们在 if 语句块内 设置了一些随后会在 for 循环中用到的变量，来取代在 if 语句块内执行完备的动作集合。我们添加了给 函数添加了几个本地变量，并且使用 printf 来格式化输出。

拓展阅读

- 《高级 Bash 脚本指南》有一章关于循环的内容，其中列举了各种各样的 for 循环实例：

  http://tldp.org/LDP/abs/html/loops1.html

- 《Bash 参考手册》描述了循环复合命令，包括了 for 循环：

  http://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs

字符串和数字

所有的计算机程序都是用来和数据打交道的。在过去的章节中，我们专注于处理文件级别的数据。 然而，许多程序问题需要使用更小的数据单位来解决，比方说字符串和数字。

在这一章中，我们将查看几个用来操作字符串和数字的 shell 功能。shell 提供了各种执行字符串操作的参数展开功能。 除了算术展开（在第七章中接触过），还有一个常见的命令行程序叫做 bc，能执行更高级别的数学运算。

参数展开

尽管参数展开在第七章中出现过，但我们并没有详尽地介绍它，因为大多数的参数展开会用在脚本中，而不是命令行中。 我们已经使用了一些形式的参数展开；例如，shell 变量。shell 提供了更多方式。

基本参数

最简单的参数展开形式反映在平常使用的变量上。例如：

$a

当 $a 展开后，会变成变量 a 所包含的值。简单参数也可能用花括号引起来：

${a}

虽然这对展开没有影响，但若该变量 a 与其它的文本相邻，可能会把 shell 搞糊涂了。在这个例子中，我们试图 创建一个文件名，通过把字符串 “_file” 附加到变量 a 的值的后面。

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

如果我们执行这个序列，没有任何输出结果，因为 shell 会试着展开一个称为 a_file 的变量，而不是 a。通过 添加花括号可以解决这个问题：

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

我们已经知道通过把数字包裹在花括号中，可以访问大于9的位置参数。例如，访问第十一个位置参数，我们可以这样做：

${11}

管理空变量的展开

几种用来处理不存在和空变量的参数展开形式。这些展开形式对于解决丢失的位置参数和给参数指定默认值的情况很方便。

${parameter:-word}

若 parameter 没有设置（例如，不存在）或者为空，展开结果是 word 的值。若 parameter 不为空，则展开结果是 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
if unset
substitute value
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

${parameter:=word}

若 parameter 没有设置或为空，展开结果是 word 的值。另外，word 的值会赋值给 parameter。 若 parameter 不为空，展开结果是 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

注意： 位置参数或其它的特殊参数不能以这种方式赋值。

${parameter:?word}

若 parameter 没有设置或为空，这种展开导致脚本带有错误退出，并且 word 的内容会发送到标准错误。若 parameter 不为空， 展开结果是 parameter 的值。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

${parameter:+word}

若 parameter 没有设置或为空，展开结果为空。若 parameter 不为空， 展开结果是 word 的值会替换掉 parameter 的值；然而，parameter 的值不会改变。

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

返回变量名的参数展开

shell 具有返回变量名的能力。这会用在一些相当独特的情况下。

${!prefix*}

${!prefix@}

这种展开会返回以 prefix 开头的已有变量名。根据 bash 文档，这两种展开形式的执行结果相同。 这里，我们列出了所有以 BASH 开头的环境变量名：

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL
BASH_VERSINFO BASH_VERSION
```

字符串展开

有大量的展开形式可用于操作字符串。其中许多展开形式尤其适用于路径名的展开。

${#parameter}

展开成由 parameter 所包含的字符串的长度。通常，parameter 是一个字符串；然而， 如果 parameter 是 @ 或者是 * 的话， 则展开结果是位置参数的个数。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

${parameter:offset}

${parameter:offset:length}

这些展开用来从 parameter 所包含的字符串中提取一部分字符。提取的字符始于 第 offset 个字符（从字符串开头算起）直到字符串的末尾，除非指定提取的长度。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

若 offset 的值为负数，则认为 offset 值是从字符串的末尾开始算起，而不是从开头。注意负数前面必须有一个空格， 为防止与 ${parameter:-word} 展开形式混淆。length，若出现，则必须不能小于零。

如果 parameter 是 @，展开结果是 length 个位置参数，从第 offset 个位置参数开始。

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

${parameter#pattern}

${parameter##pattern}

这些展开会从 paramter 所包含的字符串中清除开头一部分文本，这些字符要匹配定义的 patten。pattern 是 通配符模式，就如那些用在路径名展开中的模式。这两种形式的差异之处是该 # 形式清除最短的匹配结果， 而该 ## 模式清除最长的匹配结果。

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

${parameter%pattern}

${parameter%%pattern}

这些展开和上面的 # 和 ## 展开一样，除了它们清除的文本从 parameter 所包含字符串的末尾开始，而不是开头。

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

${parameter/pattern/string}

${parameter//pattern/string}

${parameter/#pattern/string}

${parameter/%pattern/string}

这种形式的展开对 parameter 的内容执行查找和替换操作。如果找到了匹配通配符 pattern 的文本， 则用 string 的内容替换它。在正常形式下，只有第一个匹配项会被替换掉。在该 // 形式下，所有的匹配项都会被替换掉。 该 /# 要求匹配项出现在字符串的开头，而 /% 要求匹配项出现在字符串的末尾。/string 可能会省略掉，这样会 导致删除匹配的文本。

```
[me@linuxbox~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

知道参数展开是件很好的事情。字符串操作展开可以用来替换其它常见命令比方说 sed 和 cut。 通过减少使用外部程序，展开提高了脚本的效率。举例说明，我们将修改在之前章节中讨论的 longest-word 程序， 用参数展开 ${#j} 取代命令 $(echo $j | wc -c) 及其 subshell ，像这样：

```
#!/bin/bash
# longest-word3 : find longest string in a file
for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=
        for j in $(strings $i); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
    shift
done
```

下一步，我们将使用 time 命令来比较这两个脚本版本的效率：

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)
real 0m3.618s
user 0m1.544s
sys 0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)
real 0m0.060s
user 0m0.056s
sys 0m0.008s
```

原来的脚本扫描整个文本文件需耗时3.168秒，而该新版本，使用参数展开，仅仅花费了0.06秒 —— 一个非常巨大的提高。

大小写转换

最新的 bash 版本已经支持字符串的大小写转换了。bash 有四个参数展开和 delare 命令的两个选项来支持大小写转换。

那么大小写转换对什么有好处呢？除了明显的审美价值，它在编程领域还有一个重要的角色。 让我们考虑一个数据库查询的案例。假设一个用户已经敲写了一个字符串到数据输入框中， 而我们想要在一个数据库中查找这个字符串。该用户输入的字符串有可能全是大写字母或全是小写或是两者的结合。 我们当然不希望把每个可能的大小写拼写排列填充到我们的数据库中。那怎么办？

解决这个问题的常见方法是规范化用户输入。也就是，在我们试图查询数据库之前，把用户的输入转换成标准化。 我们能做到这一点，通过把用户输入的字符全部转换成小写字母或大写字母，并且确保数据库中的条目 按同样的方式规范化。

这个 declare 命令可以用来把字符串规范成大写或小写字符。使用 declare 命令，我们能强制一个 变量总是包含所需的格式，无论如何赋值给它。

```
#!/bin/bash
# ul-declare: demonstrate case conversion via declare
declare -u upper
declare -l lower
if [[ $1 ]]; then
    upper="$1"
    lower="$1"
    echo $upper
    echo $lower
fi
```

在上面的脚本中，我们使用 declare 命令来创建两个变量，upper 和 lower。我们把第一个命令行参数的值（位置参数1）赋给 每一个变量，然后把变量值在屏幕上显示出来：

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

正如我们所看到的，命令行参数（“aBc”）已经规范化了。

有四个参数展开，可以执行大小写转换操作：

表 34-1: 大小写转换参数展开

格式	结果
${parameter,,}	把 parameter 的值全部展开成小写字母。
${parameter,}	仅仅把 parameter 的第一个字符展开成小写字母。
${parameter^^}	把 parameter 的值全部转换成大写字母。
${parameter^}	仅仅把 parameter 的第一个字符转换成大写字母（首字母大写）。

这里是一个脚本，演示了这些展开格式：

```
#!/bin/bash
# ul-param - demonstrate case conversion via parameter expansion
if [[ $1 ]]; then
    echo ${1,,}
    echo ${1,}
    echo ${1^^}
    echo ${1^}
fi
```

这里是脚本运行后的结果：

```
[me@linuxbox ~]$ ul-param aBc
abc
aBc
ABC
ABc
```

再次，我们处理了第一个命令行参数，输出了由参数展开支持的四种变体。尽管这个脚本使用了第一个位置参数， 但参数可以是任意字符串，变量，或字符串表达式。

算术求值和展开

我们在第七章中已经接触过算术展开了。它被用来对整数执行各种算术运算。它的基本格式是：

$((expression))

这里的 expression 是一个有效的算术表达式。

这个与复合命令 (( )) 有关，此命令用做算术求值（真测试），我们在第27章中遇到过。

在之前的章节中，我们看到过一些类型的表达式和运算符。这里，我们将看到一个更完整的列表。

数基

回到第9章，我们看过八进制（以8为底）和十六进制（以16为底）的数字。在算术表达式中，shell 支持任意进制的整形常量。

表 34-2: 指定不同的数基

表示法	描述
number	默认情况下，没有任何表示法的数字被看做是十进制数（以10为底）。
0number	在算术表达式中，以零开头的数字被认为是八进制数。
0xnumber	十六进制表示法
base#number	number 以 base 为底

一些例子：

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

在上面的示例中，我们打印出十六进制数 ff（最大的两位数）的值和最大的八位二进制数（以2为底）。

一元运算符

有两个二元运算符，+ 和 -，它们被分别用来表示一个数字是正数还是负数。例如，-5。

简单算术

下表中列出了普通算术运算符：

表 34-3: 算术运算符

运算符	描述
+	加
-	减
	乘
/	整除
*	乘方
%	取模（余数）

其中大部分运算符是不言自明的，但是整除和取模运算符需要进一步解释一下。

因为 shell 算术只操作整形，所以除法运算的结果总是整数：

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

这使得确定除法运算的余数更为重要：

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

通过使用除法和取模运算符，我们能够确定5除以2得数是2，余数是1。

在循环中计算余数是很有用处的。在循环执行期间，它允许某一个操作在指定的间隔内执行。在下面的例子中， 我们显示一行数字，并高亮显示5的倍数：

```
#!/bin/bash
# modulo : demonstrate the modulo operator
for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
printf "\n"
```

当脚本执行后，输出结果看起来像这样：

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

赋值运算符

尽管它的使用不是那么明显，算术表达式可能执行赋值运算。虽然在不同的上下文中，我们已经执行了许多次赋值运算。 每次我们给变量一个值，我们就执行了一次赋值运算。我们也能在算术表达式中执行赋值运算：

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

在上面的例子中，首先我们给变量 foo 赋了一个空值，然后验证 foo 的确为空。下一步，我们执行一个 if 复合命令 ((foo = 5))。 这个过程完成两件有意思的事情：1）它把5赋值给变量 foo，2）它计算测试条件为真，因为 foo 的值非零。

注意： 记住上面表达式中 = 符号的真正含义非常重要。单个 = 运算符执行赋值运算。foo = 5 是说"使得 foo 等于5"， 而 == 运算符计算等价性。foo == 5 是说"是否 foo 等于5？"。这会让人感到非常迷惑，因为 test 命令接受单个 = 运算符 来测试字符串等价性。这也是使用更现代的 [[ ]] 和 (( )) 复合命令来代替 test 命令的另一个原因。

除了 = 运算符，shell 也提供了其它一些表示法，来执行一些非常有用的赋值运算：

表34-4: 赋值运算符

表示法	描述
parameter = value	简单赋值。给 parameter 赋值。
parameter += value	加。等价于 parameter = parameter + value。
parameter -= value	减。等价于 parameter = parameter – value。
parameter = *value*	乘。等价于 parameter = parameter value。
parameter /= value	整除。等价于 parameter = parameter / value。
parameter %= value	取模。等价于 parameter = parameter % value。
parameter++	后缀自增变量。等价于 parameter = parameter + 1 (但，要看下面的讨论)。
parameter--	后缀自减变量。等价于 parameter = parameter - 1。
++parameter	前缀自增变量。等价于 parameter = parameter + 1。
--parameter	前缀自减变量。等价于 parameter = parameter - 1。

这些赋值运算符为许多常见算术任务提供了快捷方式。特别关注一下自增（++）和自减（--）运算符，它们会把它们的参数值加1或减1。 这种风格的表示法取自C 编程语言并且被其它几种编程语言吸收，包括 bash。

自增和自减运算符可能会出现在参数的前面或者后面。然而它们都是把参数值加1或减1，这两个位置有个微小的差异。 若运算符放置在参数的前面，参数值会在参数返回之前增加（或减少）。若放置在后面，则运算会在参数返回之后执行。 这相当奇怪，但这是它预期的行为。这里是个演示的例子：

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

如果我们把1赋值给变量 foo，然后通过把自增运算符 ++ 放到参数名 foo 之后来增加它，foo 返回1。 然而，如果我们第二次查看变量 foo 的值，我们看到它的值增加了1。若我们把 ++ 运算符放到参数 foo 之前， 我们得到更期望的行为：

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

对于大多数 shell 应用来说，前缀运算符最有用。

自增 ++ 和 自减 -- 运算符经常和循环操作结合使用。我们将改进我们的 modulo 脚本，让代码更紧凑些：

```
#!/bin/bash
# modulo2 : demonstrate the modulo operator
for ((i = 0; i <= 20; ++i )); do
    if (((i % 5) == 0 )); then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
printf "\n"
```

位运算符

位运算符是一类以不寻常的方式操作数字的运算符。这些运算符工作在位级别的数字。它们被用在某类底层的任务中， 经常涉及到设置或读取位标志。

表34-5：位运算符

运算符	描述
~	按位取反。对一个数字所有位取反。
<<	位左移. 把一个数字的所有位向左移动。
>>	位右移. 把一个数字的所有位向右移动。
&	位与。对两个数字的所有位执行一个 AND 操作。
|	位或。对两个数字的所有位执行一个 OR 操作。
^	位异或。对两个数字的所有位执行一个异或操作。

注意除了按位取反运算符之外，其它所有位运算符都有相对应的赋值运算符（例如，<\<=）。

这里我们将演示产生2的幂列表的操作，使用位左移运算符：

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```

逻辑运算符

正如我们在第27章中所看到的，复合命令 (( )) 支持各种各样的比较运算符。还有一些可以用来计算逻辑运算。 这里是比较运算符的完整列表：

表34-6：比较运算符

运算符	描述
<=	小于或相等
>=	大于或相等
<	小于
>	大于
==	相等

!=	不相等
&&	逻辑与
||	逻辑或
expr1? expr2:expr3	条件（三元）运算符。若表达式 expr1 的计算结果为非零值（算术真），则 执行表达式 expr2，否则执行表达式 expr3。

当表达式用于逻辑运算时，表达式遵循算术逻辑规则；也就是，表达式的计算结果是零，则认为假，而非零表达式认为真。该 (( )) 复合命令把结果映射成 shell 正常的退出码：

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

最陌生的逻辑运算符就是这个三元运算符了。这个运算符（仿照于 C 编程语言里的三元运算符）执行一个单独的逻辑测试。它用起来类似于 if/then/else 语句。它操作三个算术表达式（字符串不会起作用），并且若第一个表达式为真（或非零），则执行第二个表达式。否则，执行第三个表达式。我们可以在命令行中实验一下：

```
[me@linuxbox~]$ a=0
[me@linuxbox~]$ ((a<1?++a:--a))
[me@linuxbox~]$ echo $a
1
[me@linuxbox~]$ ((a<1?++a:--a))
[me@linuxbox~]$ echo $a
0
```

这里我们看到一个实际使用的三元运算符。这个例子实现了一个切换。每次运算符执行的时候，变量 a 的值从零变为1，或反之亦然。

请注意在表达式内执行赋值却并非易事。

当企图这样做时，bash 会声明一个错误：

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token is "-=1")
```

通过把赋值表达式用括号括起来，可以解决这个错误：

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

下一步，我们看一个使用算术运算符更完备的例子，该示例产生一个简单的数字表格：

```
#!/bin/bash
# arith-loop: script to demonstrate arithmetic operators
finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"
until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" $a $b $c
    ((a<10?++a:(finished=1)))
done
```

在这个脚本中，我们基于变量 finished 的值实现了一个 until 循环。首先，把变量 finished 的值设为零（算术假），继续执

行循环之道它的值变为非零。在循环体内，我们计算计数器 a 的平方和立方。在循环末尾，计算计数器变量 a 的值。 若它小于10（最大迭代次数），则 a 的值加1，否则给变量 finished 赋值为1，使得变量 finished 算术为真， 从而终止循环。运行该脚本得到这样的结果：

```
[me@linuxbox ~]$ arith-loop
a    a**2     a**3
=    ====     ====
0    0        0
1    1        1
2    4        8
3    9        27
4    16       64
5    25       125
6    36       216
7    49       343
8    64       512
9    81       729
10   100      1000
```

bc — 一种高精度计算器语言

我们已经看到 shell 是可以处理所有类型的整形算术的，但是如果我们需要执行更高级的数学运算或仅使用浮点数，该怎么办？ 答案是，我们不能这样做。至少不能直接用 shell 完成此类运算。为此，我们需要使用外部程序。 有几种途径可供我们采用。嵌入的 Perl 或者 AWK 程序是一种可能的方案，但是不幸的是，超出了本书的内容大纲。 另一种方式就是使用一种专业的计算器程序。这样一个程序叫做 bc，在大多数 Linux 系统中都可以找到。

该 bc 程序读取一个用它自己的类似于 C 语言的语法编写的脚本文件。一个 bc 脚本可能是一个分离的文件或者是读取 标准输入。bc 语言支持相当少的功能，包括变量，循环和程序员定义的函数。这里我们不会讨论整个 bc 语言， 仅仅体验一下。查看 bc 的手册页，其文档整理非常好。

让我们从一个简单的例子开始。我们将编写一个 bc 脚本来执行2加2运算：

```
/* A very simple bc script */
2 + 2
```

脚本的第一行是一行注释。bc 使用和 C 编程语言一样的注释语法。注释，可能会跨越多行，开始于 `/*` 结束于 `*/`。

使用 bc

如果我们把上面的 bc 脚本保存为 foo.bc，然后我们就能这样运行它：

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

如果我们仔细观察，我们看到算术结果在最底部，版权信息之后。可以通过 -q（quiet）选项禁止这些版权信息。 bc 也能够交互使用：

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

当使用 bc 交互模式时，我们简单地输入我们希望执行的运算，结果就立即显示出来。bc 的 quit 命令结束交互会话。

也可能通过标准输入把一个脚本传递给 bc 程序：

```
[me@linuxbox ~]$ bc < foo.bc
4
```

这种接受标准输入的能力，意味着我们可以使用 here 文档，here字符串，和管道来传递脚本。这里是一个使用 here 字符串的例子：

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

一个脚本实例

作为一个真实世界的例子，我们将构建一个脚本，用于计算每月的还贷金额。在下面的脚本中， 我们使用了 here 文档把一个脚本传递给 bc：

```
#!/bin/bash
# loan-calc : script to calculate monthly loan payments
PROGNAME=$(basename $0)
usage () {
    cat <<- EOF
    Usage: $PROGNAME PRINCIPAL INTEREST MONTHS
    Where:
    PRINCIPAL is the amount of the loan.
    INTEREST is the APR as a number (7% = 0.07).
    MONTHS is the length of the loan's term.
    EOF
}
if (($# != 3)); then
    usage
    exit 1
fi
principal=$1
interest=$2
months=$3
bc <<- EOF
    scale = 10
    i = $interest / 12
    p = $principal
    n = $months
    a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
    print a, "\n"
EOF
```

当脚本执行后，输出结果像这样：

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
475
1270.7222490000
```

若贷款 135,000 美金，年利率为 7.75%，借贷180个月（15年）， 这个例子计算出每月需要还贷的金额。 注意这个答案的精确度。这是由脚本中变量 scale 的值决定的。bc 的手册页提供了对 bc 脚本语言的详尽描述。 虽然 bc 的数学符号与 shell 的略有差异（bc 与 C 更相近）， 但是基于目前我们所学的内容， 大多数符号是我们相当熟悉的。

总结

在这一章中，我们学习了很多小东西，在脚本中这些小零碎可以完成“真正的工作”。随着我们编写脚本经验的增加， 能够有效地操作字符串和数字的能力将具有极为重要的价值。我们的 loan-calc 脚本表明， 甚至可以创建简单的脚本来完成一些真正有用的事情。

额外加分

虽然该 loan-calc 脚本的基本功能已经很到位了，但脚本还远远不够完善。为了额外加分，试着 给脚本 loan-calc 添加以下功能：

- 完整的命令行参数验证
- 用一个命令行选项来实现“交互”模式，提示用户输入本金、利率和贷款期限
- 输出格式美化

拓展阅读

- 《Bash Hackers Wiki》对参数展开有一个很好的论述：

  http://wiki.bash-hackers.org/syntax/pe

- 《Bash 参考手册》也介绍了这个：

  http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion

- Wikipedia 上面有一篇很好的文章描述了位运算：

  http://en.wikipedia.org/wiki/Bit_operation

- 和一篇关于三元运算的文章：

  http://en.wikipedia.org/wiki/Ternary_operation

- 还有一个对计算还贷金额公式的描述，我们的 loan-calc 脚本中用到了这个公式：

  http://en.wikipedia.org/wiki/Amortization_calculator

数组

在上一章中，我们查看了 shell 怎样操作字符串和数字的。目前我们所见到的数据类型在计算机科学圈里被 成为标量变量；也就是说，只能包含一个值的变量。

在本章中，我们将看看另一种数据结构叫做数组，数组能存放多个值。数组几乎是所有编程语言的一个特性。 shell 也支持它们，尽管以一个相当有限的形式。即便如此，为解决编程问题，它们是非常有用的。

什么是数组？

数组是一次能存放多个数据的变量。数组的组织结构就像一张表。我们拿电子表格举例。一张电子表格就像是一个 二维数组。它既有行也有列，并且电子表格中的一个单元格，可以通过单元格所在的行和列的地址定位它的位置。 数组行为也是如此。数组有单元格，被称为元素，而且每个元素会包含数据。 使用一个称为索引或下标的地址可以访问一个单独的数组元素。

大多数编程语言支持多维数组。一个电子表格就是一个多维数组的例子，它有两个维度，宽度和高度。 许多语言支持任意维度的数组，虽然二维和三维数组可能是最常用的。

Bash 中的数组仅限制为单一维度。我们可以把它们看作是只有一列的电子表格。尽管有这种局限，但是有许多应用使用它们。 对数组的支持第一次出现在 bash 版本2中。原来的 Unix shell 程序，sh，根本就不支持数组。

创建一个数组

数组变量就像其它 bash 变量一样命名，当被访问的时候，它们会被自动地创建。这里是一个例子：

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

这里我们看到一个赋值并访问数组元素的例子。通过第一个命令，把数组 a 的元素1赋值为 “foo”。 第二个命令显示存储在元素1中的值。在第二个命令中使用花括号是必需的， 以便防止 shell 试图对数组元素名执行路径名展开操作。

也可以用 declare 命令创建一个数组：

```
[me@linuxbox ~]$ declare -a a
```

使用 -a 选项，declare 命令的这个例子创建了数组 a。

数组赋值

有两种方式可以给数组赋值。单个值赋值使用以下语法：

```
name[subscript]=value
```

这里的 name 是数组的名字，subscript 是一个大于或等于零的整数（或算术表达式）。注意数组第一个元素的下标是0， 而不是1。数组元素的值可以是一个字符串或整数。

多个值赋值使用下面的语法：

```
name=(value1 value2 ...)
```

这里的 name 是数组的名字，value... 是要按照顺序赋给数组的值，从元素0开始。例如，如果我们希望 把星期几的英文简写赋值给数组 days，我们可以这样做：

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

还可以通过指定下标，把值赋给数组中的特定元素：

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

访问数组元素

那么数组对什么有好处呢？就像许多数据管理任务一样，可以用电子表格程序来完成，许多编程任务则可以用数组完成。

让我们考虑一个简单的数据收集和展示的例子。我们将构建一个脚本，用来检查一个特定目录中文件的修改次数。从这些数据中，我们的脚本将输出一张表，显示这些文件最后是在一天中的哪个小时被修改的。这样一个脚本 可以被用来确定什么时段一个系统最活跃。这个脚本，称为 hours，输出这样的结果：

```
[me@linuxbox ~]$ hours .
Hour Files Hour Files
---- ----- ---- ----
00   0     12   11
01   1     13   7
02   0     14   1
03   0     15   7
04   1     16   6
04   1     17   5
06   6     18   4
07   3     19   4
08   1     20   1
09   14    21   0
10   2     22   0
11   5     23   0
Total files = 80
```

当执行该 hours 程序时，指定当前目录作为目标目录。它打印出一张表显示一天（0-23小时）每小时内， 有多少文件做了最后修改。程序代码如下所示：

```
#!/bin/bash
# hours : script to count files by modification time
usage () {
    echo "usage: $(basename $0) directory" >&2
}
# Check that argument is a directory
if [[ ! -d $1 ]]; then
    usage
    exit 1
fi
# Initialize array
for i in {0..23}; do hours[i]=0; done
# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
    j=${i/#0}
    ((++hours[j]))
    ((++count))
done
# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count
```

这个脚本由一个函数（名为 usage），和一个分为四个区块的主体组成。在第一部分，我们检查是否有一个命令行参数， 且该参数为目录。如果不是目录，会显示脚本使用信息并退出。

第二部分初始化一个名为 hours 的数组。给每一个数组元素赋值一个0。虽然没有特殊需要在使用之前准备数组，但是 我们的脚本需要确保没有元素是空值。注意这个循环构建方式很有趣。通过使用花括号展开（{0..23}），我们能 很容易为 for 命令产生一系列的数据（words）。

接下来的一部分收集数据，对目录中的每一个文件运行 stat 程序。我们使用 cut 命令从结果中抽取两位数字的小时字段。 在循环里面，我们需要把小时字段开头的零清除掉，因为 shell 将试图（最终会失败）把从 “00” 到 “09” 的数值解释为八进制（见表34-1）。 下一步，我们以小时为数组索引，来增加其对应的数组元素的值。最后，我们增加一个计数器的值（count），记录目录中总共的文件数目。

脚本的最后一部分显示数组中的内容。我们首先输出两行标题，然后进入一个循环产生两栏输出。最后，输出总共的文件数目。

数组操作

有许多常见的数组操作。比方说删除数组，确定数组大小，排序，等等。有许多脚本应用程序。

输出整个数组的内容

下标 * 和 @ 可以被用来访问数组中的每一个元素。与位置参数一样，@ 表示法在两者之中更有用处。 这里是一个演示：

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

我们创建了数组 animals，并把三个含有两个字的字符串赋值给数组。然后我们执行四个循环看一下对数组内容进行分词的效果。 表示法 ${animals[*]} 和 ${animals[@]}的行为是一致的直到它们被用引号引起来。

确定数组元素个数

使用参数展开，我们能够确定数组元素的个数，与计算字符串长度的方式几乎相同。这里是一个例子:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]} # length of element 100
3
```

我们创建了数组 a，并把字符串 “foo” 赋值给数组元素100。下一步，我们使用参数展开来检查数组的长度，使用 @ 表示法。 最后，我们查看了包含字符串 “foo” 的数组元素 100 的长度。有趣的是，尽管我们把字符串赋值给数组元素100， bash 仅仅报告数组中有一个元素。这不同于一些其它语言的行为，数组中未使用的元素（元素0-99）会初始化为空值， 并把它们计入数组长度。

找到数组使用的下标

因为 bash 允许赋值的数组下标包含“间隔”，有时候确定哪个元素真正存在是很有用的。为做到这一点， 可以使用以下形式的参数展开：

${!array[*]}

${!array[@]}

这里的 array 是一个数组变量的名字。和其它使用符号 * 和 @ 的展开一样，用引号引起来的 @ 格式是最有用的， 因为它能展开成分离的词。

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
b
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
2
4
6
```

在数组末尾添加元素

如果我们需要在数组末尾附加数据，那么知道数组中元素的个数是没用的，因为通过 * 和 @ 表示法返回的数值不能 告诉我们使用的最大数组索引。幸运地是，shell 为我们提供了一种解决方案。通过使用 += 赋值运算符， 我们能够自动地把值附加到数组末尾。这里，我们把三个值赋给数组 foo，然后附加另外三个。

```
[me@linuxbox~]$ foo=(a b c)
[me@linuxbox~]$ echo ${foo[@]}
a b c
[me@linuxbox~]$ foo+=(d e f)
[me@linuxbox~]$ echo ${foo[@]}
a b c d e f
```

数组排序

就像电子表格，经常有必要对一列数据进行排序。Shell 没有这样做的直接方法，但是通过一点儿代码，并不难实现。

```
#!/bin/bash
# array-sort : Sort an array
a=(f e d c b a)
echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array: ${a_sorted[@]}"
```

当执行之后，脚本产生这样的结果：

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:
a b c d e f
```

脚本运行成功，通过使用一个复杂的命令替换把原来的数组（a）中的内容复制到第二个数组（a_sorted）中。 通过修改管道线的设计，这个基本技巧可以用来对数组执行各种各样的操作。

删除数组

删除一个数组，使用 unset 命令：

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}
[me@linuxbox ~]$
```

也可以使用 unset 命令删除单个的数组元素：

```
[me@linuxbox~]$ foo=(a b c d e f)
[me@linuxbox~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox~]$ unset 'foo[2]'
[me@linuxbox~]$ echo ${foo[@]}
a b d e f
```

在这个例子中，我们删除了数组中的第三个元素，下标为2。记住，数组下标开始于0，而不是1！也要注意数组元素必须 用引号引起来为的是防止 shell 执行路径名展开操作。

有趣地是，给一个数组赋空值不会清空数组内容：

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

任何引用一个不带下标的数组变量，则指的是数组元素0：

```
[me@linuxbox~]$ foo=(a b c d e f)
[me@linuxbox~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox~]$ foo=A
[me@linuxbox~]$ echo ${foo[@]}
A b c d e f
```

关联数组

现在最新的 bash 版本支持关联数组了。关联数组使用字符串而不是整数作为数组索引。 这种功能给出了一种有趣的新方法来管理数据。例如，我们可以创建一个叫做 “colors” 的数组，并用颜色名字作为索引。

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

不同于整数索引的数组，仅仅引用它们就能创建数组，关联数组必须用带有 -A 选项的 declare 命令创建。

访问关联数组元素的方式几乎与整数索引数组相同：

```
echo ${colors["blue"]}
```

在下一章中，我们将看一个脚本，很好地利用关联数组，生产出了一个有意思的报告。

总结

如果我们在 bash 手册页中搜索单词 “array”的话，我们能找到许多 bash 在哪里会使用数组变量的实例。其中大部分相当晦

涩难懂， 但是它们可能在一些特殊场合提供临时的工具。事实上，在 shell 编程中，整套数组规则利用率相当低，很大程度上归咎于这样的事实， 传统 Unix shell 程序（比如说 sh）缺乏对数组的支持。这样缺乏人气是不幸的，因为数组广泛应用于其它编程语言， 并为解决各种各样的编程问题，提供了一个强大的工具。

数组和循环有一种天然的姻亲关系，它们经常被一起使用。该

for ((expr; expr; expr))

形式的循环尤其适合计算数组下标。

拓展阅读

- Wikipedia 上面有两篇关于在本章提到的数据结构的文章：

  http://en.wikipedia.org/wiki/Scalar_(computing)

  http://en.wikipedia.org/wiki/Associative_array

奇珍异宝

在我们 bash 学习旅程中的最后一站，我们将看一些零星的知识点。当然我们在之前的章节中已经 涵盖了很多方面，但是还有许多 bash 特性我们没有涉及到。其中大部分特性相当晦涩，主要对 那些把 bash 集成到 Linux 发行版的程序有用处。然而还有一些特性，虽然不常用， 但是对某些程序问题是很有帮助的。我们将在这里介绍它们。

组命令和子 shell

bash 允许把命令组合在一起。可以通过两种方式完成；要么用一个 group 命令，要么用一个子 shell。 这里是每种方式的语法示例：

组命令：

{ command1; command2; [command3; ...] }

子 shell：

(command1; command2; [command3;...])

这两种形式的不同之处在于，组命令用花括号把它的命令包裹起来，而子 shell 用括号。值得注意的是，鉴于 bash 实现组命令的方式， 花括号与命令之间必须有一个空格，并且最后一个命令必须用一个分号或者一个换行符终止。

那么组命令和子 shell 命令对什么有好处呢？ 尽管它们有一个很重要的差异（我们马上会接触到），但它们都是用来管理重定向的。 让我们考虑一个对多个命令执行重定向的脚本片段。

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

这些代码相当简洁明了。三个命令的输出都重定向到一个名为 output.txt 的文件中。 使用一个组命令，我们可以重新编 写这些代码，如下所示：

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

使用一个子 shell 是相似的：

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

使用这样的技术，我们为我们自己节省了一些打字时间，但是组命令和子 shell 真正闪光的地方是与管道线相结合。 当构建一个管道线命令的时候，通常把几个命令的输出结果合并成一个流是很有用的。 组命令和子 shell 使这种操作变得很简单：

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

这里我们已经把我们的三个命令的输出结果合并在一起，并把它们用管道输送给命令 lpr 的输入，以便产生一个打印报告。

在下面的脚本中，我们将使用组命令，看几个与关联数组结合使用的编程技巧。这个脚本，称为 array-2，当给定一个目录名，打印出目录中的文件列表， 伴随着每个文件的文件所有者和组所有者。在文件列表的末尾，脚本打印出属于每个所有者和组的文件数目。 这里我们看到的结果（缩短的，为简单起见），是给定脚本的目录为 /usr/bin 的时候：

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6                 root        root
/usr/bin/2to3                     root        root
/usr/bin/a2p                      root        root
/usr/bin/abrowser                 root        root
/usr/bin/aconnect                 root        root
/usr/bin/acpi_fakekey             root        root
/usr/bin/acpi_listen              root        root
/usr/bin/add-apt-repository       root        root
.
.
.
/usr/bin/zipgrep                  root        root
/usr/bin/zipinfo                  root        root
/usr/bin/zipnote                  root        root
/usr/bin/zip                      root        root
/usr/bin/zipsplit                 root        root
/usr/bin/zjsdecode                root        root
/usr/bin/zsoelim                  root        root

File owners:
daemon  : 1 file(s)
root    : 1394 file(s) File group owners:
crontab : 1 file(s)
daemon  : 1 file(s)
lpadmin : 1 file(s)
mail    : 4 file(s)
mlocate : 1 file(s)
root    : 1380 file(s)
shadow  : 2 file(s)
ssh     : 1 file(s)
tty     : 2 file(s)
utmp    : 2 file(s)
```

这里是脚本代码列表（带有行号）：

```
1    #!/bin/bash
2
3    # array-2: Use arrays to tally file owners
4
5    declare -A files file_group file_owner groups owners
6
7    if [[ ! -d "$1" ]]; then
8      echo "Usage: array-2 dir" >&2
9      exit 1
10   fi
11
12   for i in "$1"/*; do
13     owner=$(stat -c %U "$i")
14     group=$(stat -c %G "$i")
15      files["$i"]="$i"
16      file_owner["$i"]=$owner
17      file_group["$i"]=$group
18      ((++owners[$owner]))
19      ((++groups[$group]))
20   done
21
22   # List the collected files
23   { for i in "${files[@]}"; do
24   printf "%-40s %-10s %-10s\n" \
25   "$i" ${file_owner["$i"]} ${file_group["$i"]}
26   done } | sort
27   echo
28
29  # List owners
30   echo "File owners:"
31   { for i in "${!owners[@]}"; do
32   printf "%-10s: %5d file(s)\n" "$i" ${owners["$i"]}
33   done } | sort
34   echo
35
36   # List groups
37   echo "File group owners:"
38   { for i in "${!groups[@]}"; do
39   printf "%-10s: %5d file(s)\n" "$i" ${groups["$i"]}
40   done } | sort
```

让我们看一下这个脚本的运行机制：

行5： 关联数组必须用带有 -A 选项的 declare 命令创建。在这个脚本中我们创建了如下五个数组：

files 包含了目录中文件的名字，按文件名索引

file_group 包含了每个文件的组所有者，按文件名索引

file_owner 包含了每个文件的所有者，按文件名索引

groups 包含了属于索引的组的文件数目

owners 包含了属于索引的所有者的文件数目

行7-10：查看是否一个有效的目录名作为位置参数传递给程序。如果不是，就会显示一条使用信息，并且脚本退出，退出状态为1。

行12-20：循环遍历目录中的所有文件。使用 stat 命令，行13和行14抽取文件所有者和组所有者， 并把值赋给它们各自的数组（行16，17），使用文件名作为数组索引。同样地，文件名自身也赋值给 files 数组。

行18-19：属于文件所有者和组所有者的文件总数各自加1。

行22-27：输出文件列表。为做到这一点，使用了 “${array[@]}” 参数展开，展开成整个的数组元素列表， 并且每个元素被当做是一个单独的词。从而允许文件名包含空格的情况。也要注意到整个循环是包裹在花括号中， 从而形成了一个组命令。这样就允许整个循环输出会被管道输送给 sort 命令的输入。这是必要的，因为 展开的数组元素是无序的。

行29-40：这两个循环与文件列表循环相似，除了它们使用 “${!array[@]}” 展开，展开成数组索引的列表 而不是数组元素的。

进程替换

虽然组命令和子 shell 看起来相似，并且它们都能用来在重定向中合并流，但是两者之间有一个很重要的不同。 然而，一个组命令在当前 shell 中执行它的所有命令，而一个子 shell（顾名思义）在当前 shell 的一个 子副本中执行它的命令。这意味着运行环境被复制给了一个新的 shell 实例。当这个子 shell 退出时，环境副本会消失， 所以在子 shell 环境（包括变量赋值）中的任何更改也会消失。因此，在大多数情况下，除非脚本要求一个子 shell， 组命令比子 shell 更受欢迎。组命令运行很快并且占用的内存也少。

我们在第20章中看到过一个子 shell 运行环境问题的例子，当我们发现管道线中的一个 read 命令 不按我们所期望的那样工作的时候。为了重现问题，我们构建一个像这样的管道线：

```
echo "foo" | read
echo $REPLY
```

该 REPLY 变量的内容总是为空，是因为这个 read 命令在一个子 shell 中执行，所以它的 REPLY 副本会被毁掉， 当该子 shell 终止的时候。因为管道线中的命令总是在子 shell 中执行，任何给变量赋值的命令都会遭遇这样的问题。 幸运地是，shell 提供了一种奇异的展开方式，叫做进程替换，它可以用来解决这种麻烦。进程替换有两种表达方式：

一种适用于产生标准输出的进程：

```
<(list)
```

另一种适用于接受标准输入的进程：

```
>(list)
```

这里的 list 是一串命令列表：

为了解决我们的 read 命令问题，我们可以雇佣进程替换，像这样：

```
read < <(echo "foo")
echo $REPLY
```

进程替换允许我们把一个子 shell 的输出结果当作一个用于重定向的普通文件。事实上，因为它是一种展开形式，我们可以检验它的真实值：

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

通过使用 echo 命令，查看展开结果，我们看到子 shell 的输出结果，由一个名为 /dev/fd/63 的文件提供。

进程替换经常被包含 read 命令的循环用到。这里是一个 read 循环的例子，处理一个目录列表的内容，内容创建于一个子 shell：

```
#!/bin/bash
# pro-sub : demo of process substitution
while read attr links owner group size date time filename; do
    cat <<- EOF
        Filename:      $filename
        Size:          $size
        Owner:         $owner
        Group:         $group
        Modified:      $date $time
        Links:         $links
        Attributes:    $attr
    EOF
done < <(ls -l | tail -n +2)
```

这个循环对目录列表的每一个条目执行 read 命令。列表本身产生于该脚本的最后一行代码。这一行代码把从进程替换得到的输出 重定向到这个循环的标准输入。这个包含在管道线中的 tail 命令，是为了消除列表的第一行文本，这行文本是多余的。

当脚本执行后，脚本产生像这样的输出：

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename: addresses.ldif
Size: 14540
Owner: me
Group: me
Modified: 2009-04-02 11:12
Links:
1
Attributes: -rw-r--r--
Filename: bin
Size: 4096
Owner: me
Group: me
Modified: 2009-07-10 07:31
Links: 2
Attributes: drwxr-xr-x
Filename: bookmarks.html
Size: 394213
Owner: me
Group: me
```

陷阱

在第10章中，我们看到过程序是怎样响应信号的。我们也可以把这个功能添加到我们的脚本中。然而到目前为止， 我们所编写过的脚本还不需要这种功能（因为它们运行时间非常短暂，并且不创建临时文件），大且更复杂的脚本 可能会受益于一个信息处理程序。

当我们设计一个大的，复杂的脚本的时候，若脚本仍在运行时，用户注销或关闭了电脑，这时候会发生什么，考虑到这一点

非常重要。 当像这样的事情发生了，一个信号将会发送给所有受到影响的进程。依次地，代表这些进程的程序会执行相应的动作，来确保程序 合理有序的终止。比方说，例如，我们编写了一个会在执行时创建临时文件的脚本。在一个好的设计流程，我们应该让脚本删除创建的 临时文件，当脚本完成它的任务之后。若脚本接收到一个信号，表明该程序即将提前终止的信号， 此时让脚本删除创建的临时文件，也会是很精巧的设计。

为满足这样需求，bash 提供了一种机制，众所周知的 trap。陷阱由被恰当命令的内部命令 trap 实现。 trap 使用如下语法：

trap argument signal [signal...]

这里的 argument 是一个字符串，它被读取并被当作一个命令，signal 是一个信号的说明，它会触发执行所要解释的命令。

这里是一个简单的例子：

```
#!/bin/bash
# trap-demo : simple signal handling demo
trap "echo 'I am ignoring you.'" SIGINT SIGTERM
for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

这个脚本定义一个陷阱，当脚本运行的时候，这个陷阱每当接受到一个 SIGINT 或 SIGTERM 信号时，就会执行一个 echo 命令。 当用户试图通过按下 Ctrl-c 组合键终止脚本运行的时候，该程序的执行结果看起来像这样：

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

正如我们所看到的，每次用户试图中断程序时，会打印出这条信息。

构建一个字符串形成一个有用的命令序列是很笨拙的，所以通常的做法是指定一个 shell 函数作为命令。在这个例子中， 为每一个信号指定了一个单独的 shell 函数来处理：

```
#!/bin/bash
# trap-demo2 : simple signal handling demo
exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
}
exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}
trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM
for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

这个脚本的特色是有两个 trap 命令，每个命令对应一个信号。每个 trap，依次，当接受到相应的特殊信号时， 会执行指定的 shell 函数。注意每个信号处理函数中都包含了一个 exit 命令。没有 exit 命令， 信号处理函数执行完后，该脚本将会继续执行。

当用户在这个脚本执行期间，按下 Ctrl-c 组合键的时候，输出结果看起来像这样：

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```

> 临时文件
>
> 把信号处理程序包含在脚本中的一个原因是删除临时文件，在脚本执行期间，脚本可能会创建临时文件来存放中间结果。 命名临时文件是一种艺术。传统上，在类似于 unix 系统中的程序会在 /tmp 目录下创建它们的临时文件，/tmp 是一个服务于临时文件的共享目录。然而，因为这个目录是共享的，这会引起一定的安全顾虑，尤其对那些用 超级用户特权运行的程序。除了为暴露给系统中所有用户的文件设置合适的权限，这一明显步骤之外， 给临时文件一个不可预测的文件名是很重要的。这就避免了一种为大众所知的 temp race 攻击。 一种创建一个不可预测的（但是仍有意义的）临时文件名的方法是，做一些像这样的事情：
>
> tempfile=/tmp/$(basename $0).$$.$RANDOM
>
> 这将创建一个由程序名字，程序进程的 ID（PID）文件名，和一个随机整数组成。注意，然而，该 $RANDOM shell 变量 只能返回一个范围在1-32767内的整数值，这在计算机术语中不是一个很大的范围，所以一个单一的该变量实例是不足以克服一个坚定的攻击者的。
>
> 一个比较好的方法是使用 mktemp 程序（不要和 mktemp 标准库函数相混淆）来命名和创建临时文件。 这个 mktemp 程序接受一个用于创建文件名的模板作为参数。这个模板应该包含一系列的 “X” 字符， 随后这些字符会被相应数量的随机字母和数字替换掉。一连串的 “X” 字符越长，则一连串的随机字符也就越长。 这里是一个例子：
>
> tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
>
> 这里创建了一个临时文件，并把临时文件的名字赋值给变量 tempfile。因为模板中的 “X” 字符会被随机字母和 数字代替，所以最终的文件名（在这个例子中，文件名也包含了特殊参数 $$ 的展开值，进程的 PID）可能像这样：
>
> /tmp/foobar.6593.UOZuvM6654
>
> 对于那些由普通用户操作执行的脚本，避免使用 /tmp 目录，而是在用户主目录下为临时文件创建一个目录， 通过像这样的一行代码：
>
> [[-d $HOME/tmp]] || mkdir $HOME/tmp

异步执行

有时候需要同时执行多个任务。我们已经知道现在所有的操作系统若不是多用户的但至少是多任务的。 脚本也可以构建成多任务处理的模式。

通常这涉及到启动一个脚本，依次，启动一个或多个子脚本来执行额外的任务，而父脚本继续运行。然而，当一系列脚本 以这种方式运行时，要保持父子脚本之间协调工作，会有一些问题。也就是说，若父脚本或子脚本依赖于另一方，并且 一个脚本必须等待另一个脚本结束任务之后，才能完成它自己的任务，这应该怎么办？

bash 有一个内置命令，能帮助管理诸如此类的异步执行的任务。wait 命令导致一个父脚本暂停运行，直到一个 特定的进程（例如，子脚本）运行结束。

等待

首先我们将演示一下 wait 命令的用法。为此，我们需要两个脚本，一个父脚本：

```
#!/bin/bash
# async-parent : Asynchronous execution demo (parent)
echo "Parent: starting..."
echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."
echo "Parent: continuing..."
sleep 2
echo "Parent: pausing to wait for child to finish..."
wait $pid
echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

和一个子脚本：

```
#!/bin/bash
# async-child : Asynchronous execution demo (child)
echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

在这个例子中，我们看到该子脚本是非常简单的。真正的操作通过父脚本完成。在父脚本中，子脚本被启动， 并被放置到后台运行。子脚本的进程 ID 记录在 pid 变量中，这个变量的值是 $! shell 参数的值，它总是 包含放到后台执行的最后一个任务的进程 ID 号。

父脚本继续，然后执行一个以子进程 PID 为参数的 wait 命令。这就导致父脚本暂停运行，直到子脚本退出， 意味着父脚本结束。

当执行后，父子脚本产生如下输出：

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.
```

命名管道

在大多数类似也 Unix 的操作系统中，有可能创建一种特殊类型的饿文件，叫做命名管道。命名管道用来在 两个进程之间建立连接，也可以像其它类型的文件一样使用。虽然它们不是那么流行，但是它们值得我们去了解。

有一种常见的编程架构，叫做客户端-服务器，它可以利用像命名管道这样的通信方式， 也可以使用其它类型的进程间通信方式，比如网络连接。

最为广泛使用的客户端-服务器系统类型是，当然，一个 web 浏览器与一个 web 服务器之间进行通信。 web 浏览器作为客户端，向服务器发出请求，服务器响应请求，并把对应的网页发送给浏览器。

命令管道的行为类似于文件，但实际上形成了先入先出（FIFO）的缓冲。和普通（未命令的）管道一样， 数据从一端进入，然后从另一端出现。通过命令管道，有可能像这样设置一些东西：

```
process1 > named_pipe
```

和

```
process2 < named_pipe
```

表现出来就像这样：

```
process1 | process2
```

设置一个命名管道

首先，我们必须创建一个命名管道。使用 mkfifo 命令能够创建命令管道：

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me
me
0 2009-07-17 06:41 pipe1
```

这里我们使用 mkfifo 创建了一个名为 pipe1 的命名管道。使用 ls 命令，我们查看这个文件， 看到位于属性字段的第一个字母是 “p”，表明它是一个命名管道。

使用命名管道

为了演示命名管道是如何工作的，我们将需要两个终端窗口（或用两个虚拟控制台代替）。 在第一个终端中，我们输入一个简单命令，并把命令的输出重定向到命名管道：

```
[me@linuxbox ~]$ ls -l > pipe1
```

我们按下 Enter 按键之后，命令将会挂起。这是因为在管道的另一端没有任何接受数据。当这种现象发生的时候， 据说是管道阻塞了。一旦我们绑定一个进程到管道的另一端，该进程开始从管道中读取输入的时候，这种情况会消失。 使用第二个终端窗口，我们输入这个命令：

```
[me@linuxbox ~]$ cat < pipe1
```

然后产自第一个终端窗口的目录列表出现在第二个终端中，并作为来自 cat 命令的输出。在第一个终端 窗口中的 ls 命令一旦它不再阻塞，会成功地结束。

总结

嗯，我们已经完成了我们的旅程。现在剩下的唯一要做的事就是练习，练习，再练习。 纵然在我们的长途跋涉中，我们涉及了很多命令，但是就命令行而言，我们只是触及了它的表面。 仍留有成千上万的命令行程序，需要去发现和享受。开始挖掘 /usr/bin 目录吧，你将会看到！

拓展阅读

- bash 手册页的 “复合命令” 部分包含了对组命令和子 shell 表示法的详尽描述。
- bash 手册也的 EXPANSION 部分包含了一小部分进程替换的内容：
- 《高级 Bash 脚本指南》也有对进程替换的讨论：

  http://tldp.org/LDP/abs/html/process-sub.html
- 《Linux 杂志》有两篇关于命令管道的好文章。第一篇，源于1997年9月：

http://www.linuxjournal.com/article/2156

- 和第二篇，源于2009年3月：

  http://www.linuxjournal.com/content/using-named-pipes-fifos-bash